A D-Box koordinációs nyelv és a futtató rendszer

***Clean* funkcionális nyelvi programok elosztott futtatásának támogatása**

Hernyák Zoltán

http://aries.ektf.hu/~hz

hz@aries.ektf.hu

Doktori értekezés

2009

Témavezető: *Dr. Horváth Zoltán, egyetemi tanár*
Eötvös Lóránd Tudományegyetem, Informatikai Kar
H-1117 Budapest, Pázmány Péter sétány 1/C.

ELTE IK Informatikai Doktori Iskola

Doktori program: Az informatika alapjai és módszertana

Az iskola és a program vezetője: Dr. Demetrovics János akadémikus

Ezt az értekezést az Eötvös Lóránd Tudományegyetem az Informatika alapjai című doktori programjának keretében készítettem 2001 és 2009 között és ezúton benyújtom az ELTE doktori PhD fokozatának elnyerése céljából.

Budapest, 2009. május 21.

.....

Hernyák Zoltán

jelölt

Tanúsítom, hogy Hernyák Zoltán doktorjelölt 2001 és 2009 között a fent megnevezett doktori program keretében irányításommal végezte munkáját. Az értekezésben foglaltak a jelölt önálló munkáján alapulnak, az eredményekhez önálló alkotó tevékenységével meghatározóan hozzájárult. Az értekezés elfogadását javaslom.

Budapest, 2009. május 21.

.....

Dr. Horváth Zoltán

egyetemi tanár, témavezető

Tartalomjegyzék

Bevezető

A számítógépek egyik kiemelt fejlesztési iránya a számítási teljesítmény fokozása. Az órajel növelése, a cache technológia fejlődése és integrálása a processzor különböző műveletvégzési fázisaiba komoly teljesítménynövekedést okozott. Az egyetlen számítógépbe több processzoregység integrálása az elmúlt években az asztali kategóriájú számítógépek esetén is elérhetővé vált.

Az irányzat szuperszámítógépek építésében teljesedik ki. A mai napig elsősorban nagy erőforrásokkal rendelkező kutatóintézetekben, kormányzati megrendelésekre folyik ilyen rendszerek tervezése és telepítése. Az IBM 2012-ben fog egy *Sequoia* kódnevű projektet [1] befejezni, amely egy 20 PetaFLOPS számítási teljesítményű számítógépfürtöt takar, melyekben 1.6 PetaByte memória lesz telepítve. Érdekesség, hogy ezen rendszerek tervezése során ma egyre fontosabb szempont az energiafelhasználás és egyéb környezetvédelmi jellemzők figyelembe vétele.

A szuperszámítógépek vásárlása, üzemeltetése költséges. Asztali teljesítményű számítógépeket azonban könnyű a környezetünkben találni. Az elosztott számítási technikák segítségével ebből előnyt lehet kovácsolni. A hálózatok elterjedésével a számítási feladatoknak a kis teljesítményű számítógépek közötti elosztása technikailag egyszerűvé vált. Az internet terjedésével, a protokollok szabvánnyá válásával a távoli számítógépek elérése, munkára fogása egyre könnyebb és gyakoribb megoldás. Projektek jelentek meg, melyek a gépek *idle* (üresjárati) idejét használták ki. A feladatot kiadó szempontjából ez a megoldás gyakorlatilag ingyenes, és nagy számításigényű feladatokat is meg lehet segítségével oldani. Az emberek egyre szívesebben ajánlják fel a gépeik szabad kapacitását tudományos célok megvalósítása érdekében. Akadályozó tényező a bizalmatlanság, a vírusoktól való félelem.

Egyszerűbb az eset, amikor a felhasználandó gépek mindegyike egy adott vállalat birtokában van, és dedikált cél egy konkrét feladat futtatása. Ezek a gépek jellemzően kis távolságra helyezkednek el egymástól, nagy sebességű (100Mb, gigabit) Ethernet hálózaton csatlakoznak egymáshoz. A projektek futtatását ütemezni kell. Az aránylag olcsó, kis teljesítményű gépek összekapcsolásával nagy számítási kapacitás érhető el. Amennyiben valamelyik gép meghibásodik, szerviz idejére nem kell leállítani a teljes rendszert, a teljesítmény kis csökkenése mellett is megőrizhető az üzemképesség.

Amennyiben a szabad számítási kapacitások már rendelkezésünkre állnak, megfelelő

programozási nyelvet kell választani a feladat implementálására. Ez sajnos nem könnyű feladat. Sok programozási nyelv támogatja az elosztott programok fejlesztését, de nagy a hiány olyan operációs rendszerbeli komponensekben, amelyek az elosztott futtatást és az elosztott fájlrendszert támogatnák.

Az elterjedt objektum orientált nyelvek, mint a Java [2] vagy a .NET nyelvek [4] saját távoli metódushívási koncepciót [3] fejlesztettek ki, melyek egymással nem kompatibilisek, és nem adnak támogatást a kód távoli gépre másolásával, illetve indításával kapcsolatosan. A Web Service [5] koncepció teljesen nyitott, nem kötődik egyetlen programozási nyelvhez sem, de abból a feltevésből indul ki, hogy a szerviz már eleve a szerver gépre van telepítve. A verzióváltás nehezen oldható meg.

A Corba [7] szabvány támogatja az OOP nyelveken történő elosztott szoftverfejlesztést, saját interface leíró nyelvvel rendelkezik, és egyes implementációi távoli objektumaktiválási lehetőségeket is támogatnak. Sajnálatos módon a Microsoft.NET kívül esik a Corba lefedettségi körén, lévén hogy nem tagja a Object Management Group csoportnak. A Microsoft saját megoldása, a COM, DCOM, .NET Remoting pedig szinte teljesen belterjes megoldásnak tekinthető. Ugyanakkor sajnálatos, hogy a Microsoft saját megoldása sem támogatja semmilyen formában a futtatható kód terjesztését, másolását még a Windows operációs rendszerek között sem.

A Linux, Unix rendszereken az ftp, scp lehetőségek használatával a kód átmásolása a távoli gépre megoldható, és ssh segítségével a kód indítása is. Ugyanakkor körülményes az egyes Linux példányok beállítása az engedélyek szempontjából. Ezek a megoldások ezen felül Linux specifikusak, nem tudják kezelni az esetleg megtalálható Windows operációs rendszert használó gépeket.

Linux operációs rendszer alatt megoldást nyújthat az MPI [8] vagy PVM [9] library használata. Ezek kommunikációs függvénygyűjteményeknek tekinthetőek, melyek üzenetküldési primitívekkel támogatják az alkalmazások közötti kommunikációt. Az alkalmazások távoli gépre juttatására semmilyen támogatást nem adnak, de azok megtalálásához már igen. A processzek azonosítását sorszámok végzik. A program írásakor ezek száma ismert kell, hogy legyen. Az üzenetküldés során adott sorszámú processz egy másik adott sorszámú processznek tud üzenetet küldeni, mely adatot is tartalmazhat. Az üzenet fejrészében egy egész számérték segítségével megjelölhető az üzenet jellege (típusa), mely segíthet a túloldalon az üzenet tartalmának megértésében és dekódolásában. Az üzenetek kezelése ettől nem válik típusbiztossá: a fogadó oldal az üzenet törzsét alkotó bájtsorozatot értelmezheti rosszul is. Ezen felül lehetővé teszik bizonyos korlátok mellett az üzenetek pufferelését, valamint biztosítják az aszinkron üzenetfeldolgozást is.

A funkcionális programozási nyelvek is egyre nagyobb támogatást adnak az elosztott programozási technikák fejlesztésének. Ezen nyelvek két fontos előnnyel bírnak. Az egyik, hogy matematikai számítások leírására kézenfekvőbb eszközök, mint az impera-

tív programozási nyelvek, ezért a számítási feladatok kódolása egyszerűbb. A másik előny, hogy a kiértékelési rendszerük szinte kínálja a párhuzamos feldolgozás lehetőségét. Sajnos ez nehezen vihető át elosztott rendszerekbe, mivel a kiértékelési gráf mélységi másolása a gépek között nem triviális feladat.

Párhuzamos kiértékelési rendszerrel bír a *Concurrent Haskell* [12, 13]. Ez mindössze négy nyelvi primitívvel bővítette a Haskell nyelvet, melyek közül a *forkIO* a legjelentősebb: új szál indítása, mely paraméterként egy Haskell kifejezést kap. A szinkronizációt mutable változókon (*MVars*) keresztül oldották meg, melyek típussal rendelkeztek, és egy időben csak egy értéket képesek tárolni. Az egyik szál az értéket elhelyezi ebbe a változóba, a másik szál kiolvassa belőle. A szinkronizálási várakozás a szálak *suspend* képességével van megoldva, elkerülve ezzel a *busy waiting*-et.

A JoCaml nyelv az Objective Caml [17] nyelv kiegészítése a *join calculus* segítségével. A JoCaml nyelv kifejezetten párhuzamos és elosztott fejlesztési támogatással rendelkezik. Ez egy nem tisztán funkcionális nyelv, mely imperatív és OOP elemeket is tartalmaz [16]. A legfontosabb két absztrakt nyelvi elem a csatorna és az absztrakt hely fogalma. A JoCaml nyelvben mobil kód írására van lehetőség, melynek itteni neve az *ágens*. A folyamatok közötti kommunikációs csatorna állandó marad, miközben az ágensek helyet változtathatnak. Az absztrakt hely fogalma magában foglalja az ágens kódjának és aktuális fizikai helyének együttesét. A helyváltoztatás után az ágensek a korábbi állapotukból folytatják a működésüket. A különböző gépeken futó folyamatok között kezdetben nincs kapcsolat, az egymásra találást névszolgáltató támogatja. A *let def* segítségével hozhatunk létre szinkron és aszinkron csatornákat.

Az ERLANG [18] az Ericson és az Ellemtel Computer Science Laboratories fejlesztése. Az Erlang egy funkcionális programozási nyelv, amely konkurrens, valós idejű, elosztott és nagy hibatűrő képességű rendszerek készítését teszi lehetővé. Az Ericsson az Erlang Open Telecom Platform kiterjesztését telekommunikációs rendszerek fejlesztésére használja. A nyelv beépített eszközökkel rendelkezik az elosztottság és az üzenetküldés területén, közös memória terület nélkül, üzenetküldésekkel valósítja meg az elosztottságot. Támogatja más nyelven írt programkomponensek integrálását, viszont a típusossága gyenge.

A *Hume* [10] erősen típusos, funkcionális alapokkal bíró programozási nyelv, első verziója 2000 novemberében jelent meg. A *Hume* elsődleges prioritása a futási idő és erőforrás-felhasználás limitáltságának megőrzése. A *Hume* öt absztrakciós szintet különböztet meg. Aszinkron kommunikációs modellt használ, melynek alapeleme a *doboz*. A dobozoknak egyedi azonosítóik vannak, be-, és kimenő adataik pedig típusosak. A dobozokat össze lehet kötni, melynek leírása a doboz definícióktól különálló módon történik meg – ennek során ellenőrzésre kerül, hogy az összekötés típushelyes-e. A dobozokat eszközökön (stream, port, stb.) keresztül is össze lehet kapcsolni. A dobozok akár saját magukhoz is köthetők, egyelemű hurkot alkotva. A dobozok között húzódó

kapcsolatot *vezeték*nek nevezhetjük. A vezetékekre indulási értékeket lehet elhelyezni, melyek hurkok kialakítása esetén hasznosak. A dobozokhoz deklarálni lehet milyen kivételeket képesek kezelni, illetve költségeket (futási idő, erőforrások) lehet rendelni.

Az Eden funkcionális programozási nyelvhez Jost Berthold készített egy *EdI* nevű implementációs nyelvet [11], melyben csatornalétrehozó, és adatküldő primitíveket definiált. Az Eden lusta kiértékelése miatt kiértékelési stratégiákat képezett, melyekkel a küldő oldalon az érték kiszámítása (és küldése) kikényszeríthető. Ugyanakkor az Eden programok képesek több szálon futni, a szálak maguk csatlakozhatnak a csatornákhoz. Az *EdI* nyelven párhuzamos kiértékelésű sablonok, párhuzamos kiértékelésű magasabb rendű függvények definiálhatók.

Az egyik jelentős kutatási irány a skeletonok használata, mint magasabb rendű függvény a számítási minták leírására [19, 20]. Egy ilyen skeleton paraméterezhető típussal, függvénnyel és kiértékelési stratégiával [24].

A Clean [34] funkcionális programozási nyelv egy nonprofit fejlesztés, mely nagyon hasonlít a Haskell nyelvhez. A hozzá fejlesztett ObjectIO library [32] kiegészítésével interaktív programok is fejleszthetők, melyek menüt, párbeszédablakokat is tartalmazhatnak. A *unique* típus lehetővé teszi hogy tisztán funkcionális megközelítésben kezeljünk erőforrásokat. Egy ilyen *unique* típusú érték nem duplikálható. Valójában a Clean IDE is ezen library segítségével készült. Sajnos ezen ObjectIO portolása Linux környezetben nem teljes, így heterogén rendszerekben nem lehet rá alapozni.

A Concurrent Clean [35, 36] egy tisztán funkcionális, erősen típusos nyelv volt, mely párhuzamos és elosztott kiértékelést is támogatott. Sajnálatos módon a nyelv ezen kiterjesztése a nyelvi verziókkal nem tartotta a lépést, fejlesztése leállt.

A Clean korai időszakában volt egy transputer támogatású nyelvi verzió is [25]. Annotációk segítségével [26] lehetett megadni, hogy a kifejezés mely részeit lehet párhuzamosan kiértékelni. A párhuzamosítási stratégiák [27] ezen annotációkon alapultak, segítségükkel kiértékelési sorrendet lehetett megadni.

Egy korai megvalósításban [28, 31] a Clean programokat direkt CORBA hívásokkal lehetett összekapcsolni, melyhez Clean-C interfészt került kifejlesztésre. Nehézkes és nehezen áttekinthető megoldás volt.

Szükségesnek látszott egy magasabb szintű processzleíró és koordinációs rendszer fejlesztése [21, 22]. E célból *Zsók Viktória*, az ELTE kutatója megtervezte a D-Clean nyelvet, amelynek segítségével a kommunikáció leírható volt funkcionális nyelvi kifejezésekhez nagyon hasonló eszközökkel [30]. Ezen elosztott számítási kifejezés először egy köztes D-Box nyelvre kerül lefordításra [30], hasonlóan a [23]-ban bemutatott ötlethez. A továbbiakban a D-Box nyelvi leírásból generálódik a futtatható kód.

A fordítás során a D-Clean nyelv speciális *DistrStart* kifejezését eltávolítjuk, a generált kód már Clean nyelvű. A számítási csomópontok közötti adatáramlást típusos csatornákon keresztül valósítjuk meg, melyeket a futtató rendszer példányosít.

A dolgozat elkészítését az motiválta, hogy a Clean jelenleg nem tartalmaz párhuzamos vagy elosztott kiértékelési rendszert, ugyanakkor a Clean nyelv egy nagyon jól tervezett, egyszerű és jól használható programozási nyelv. Az előzőekben említett megoldások alkalmazása során igyekeztünk a Clean nyelvből csak olyan elemeket felhasználni, amelyek előre láthatóan a következő verzióban is részét fogják képezni a nyelvnek, vagyis a megoldás időtállóságára hangsúlyt fektettünk. Ennek megfelelően nem végeztünk módosításokat sem a kiértékelési rendszeren (*backend*), sem az aktuális verziójú Clean compileren. Nem használtuk ki az ObjectIO platformspecifikus lehetőségeit sem, hogy a megoldás ne kötődjön sem a Linux, sem a Windows környezethez.

Célkitűzések

Céljaim a következők voltak:

1. Létrehozni egy olyan koordinációs nyelvet, amely működésében támogatja az elosztott funkcionális programozást. Ez a D-Clean magasabb absztrakciójú koordinációs nyelv támogatására készüljön, s e módon alkalmas legyen *Clean* funkcionális programozási nyelven elosztott programozási megoldások fejlesztésére. A koordinációs nyelv szintaktikai és statikus szemantikai szabályait megalkotni, melynek alapján a D-Box nyelven leírt gráfdefiníció feldolgozható és ellenőrizhető.

2. A koordinációs nyelv működésével kapcsolatos specifikációkat megadni, melynek ismeretében kódgeneráló eszköz készíthető. A kódgenerálás során külső sablon fájlokat alkalmazni, melyek akár más platform, köztes réteg esetén is kidolgozhatóak. A kódgeneráló eszköz a sablon fájlokban szereplő kódot típussal paraméterezni fel, és *Clean* programozási nyelvi forráskódokat előállítani. Megalkotni azokat a makrókat, melyeket a kódgeneráló eszköz alkalmaz, és amelyek segítségével a sablonok elkészíthetők. Létrehozni konkrét sablon fájlokat, melynek segítségével valamely operációs rendszer és köztes réteg esetén kód generálható.

3. Megalkotni a köztes réteg szolgáltatásait kiegészítő futtató rendszert, az API függvényeinek informális szintaktikai és szemantikai leírását. Ezen függvényekre a sablon fájlokban hivatkozni lehet. Ezen rendszer legyen alkalmas a D-Box nyelvi leírás alapján generált projektek futtatásának előkészítésére, futás közbeni támogatására.

4. Az előzőekben megalkotott környezetet tesztelni, konkrét elosztott számítási feladatokat futtatni, hatékonyságot, teljesítményt mérni.

Az értekezés felépítése

Az első fejezetben példákon keresztül mutatjuk be a D-Box koordinációs nyelvet, a nyelvi alapelemeket, azok három fő szerkezeti részét: az input csatornákat és protokollokat, a számítási kifejezést, és az output protokollokat. A második fejezet a D-Box koordinációs nyelv statikus szemantikai leírását tartalmazza, melyben szerepet kap az input protokollok és a számítási kifejezés argumentumainak összefüggéseivel kapcsolatos helyesség, a kifejezés által előállított értékek és az output protokoll összekapcsolásával kapcsolatos helyesség. Ezen felül a csomópontok összekapcsolását végző élek, csatornák típusainak és helyes hivatkozásainak statikus elemzése, végül az algráf indítások, és az algráfok egymás közötti kommunikációjának vizsgálata. Ezen két fejezet az első célkitűzést valósítja meg.

A harmadik fejezet tárgyalja a D-Box nyelv specifikációját, beleértve a csatornaműveleteket, a protokollok és a csatornák kapcsolatát, a leképezés megvalósítását a csatornák és a számítási kifejezések között. Ezen felül tárgyalja, hogy a csatornahivatkozások milyen működést váltanak ki a projekt indítási és futási folyamataiban. Az ötödik fejezetben a koordinációs nyelvhez készített fordító és kódgeneráló szoftver működése kerül ismertetésre, mely támaszkodik a korábban definiált szemantikai szabályokra és specifikációkra. Ugyanezen fejezet a kódgeneráláshoz használt módszereket, a külső fájlok jelentését is ismerteti. Továbbá bemutatásra kerül a generált forráskód működése, a futtató rendszerrel kapcsolatos együttműködés. Az elveket implementáló sablon fájlok a lemezmellékleten találhatók, helyhiány miatt a sablonok forráskódjai nem kerültek bele az írott anyagba. Szintén a lemezmellékletre került az elkészített D-Box fordító és kódgeneráló eszköz, melynek a C# nyelvi forráskódja is szerepel a mellékelt lemezen. Ezzel a második célkitűzésben foglaltak megvalósultak.

A harmadik célkitűzésben foglaltakban szereplő API leírásoknak a C függelék ad helyet. A futtató rendszer működését természetes műveleti szemantikai eszközzel adtuk meg, mely a negyedik fejezet témája. A futtató rendszer implementációjával kapcsolatos leírásokat az ötödik fejezet tartalmazza. A futtató rendszer, mint elosztott rendszer, a D-Box nyelv jellemzése szintén az ötödik fejezet végén található. Ezzel a harmadik célkitűzés is megvalósult.

A hatodik fejezetben egy konkrét feladat ismertetése, implementációja, a mérési környezet leírása, a mérési eredmények, a következtetések kerülnek ismertetésre. Ezen fejezet igazolja, hogy a koordinációs nyelv, a futtató rendszer, és a kidolgozott sablonok működőképes rendszerré állnak össze, mely hatékony eszközzé válhat. Ezen fejezet teljesíti a negyedik célkitűzést.

Köszönetnyilvánítás

Köszönettel tartozom elsősorban témavezetőmnek, aki példamutató volt mind emberileg, mind szakmailag a doktori folyamatom minden fázisában. Biztatott, amikor biztatni kellett, mosolygott, amikor morgolódni kellett, és mindig talált egy könyvet, amit a kezembe nyomhatott, amikor arra volt szükségem.

Hasonlóan köszönettel tartozom kutatótársamnak, Zsók Viktóriának, aki a legváratlanabb pillanatokban a legváratlanabb kérdéseket tette fel, melyekre csak első pillanatban lehetett triviális válaszokat adni.

Szintén köszönettel tartozom Diviánszky Péternek, aki elképesztő lustaságot tud varázsolni bármilyen Clean kódba, és a felkiáltójelek és csillag karakterek alkalmazását művészi szintre fejlesztette.

Köszönet illeti Tómács Tibort, aki a LaTeX tudásomat megfelelő magasságokba emelte, és döbbenetes türelemmel javítgatta és stílusozta ezt az művet is.

Elismerés és tisztelet illeti Csőke Lajos és Rimán János tanár urakat, akik nem csak korábbiakban mutattak nekem példát szakmailag és emberileg, de ezen disszertácó elkészítése során is sokat segítettek tanácsaikkal, javaslataikkal.

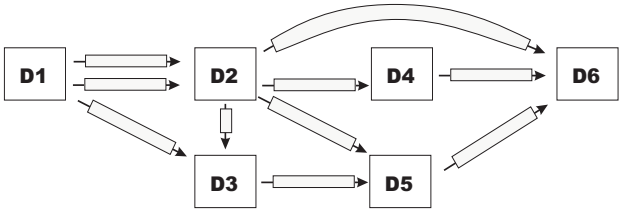
1. fejezet

A D-Box nyelv

Ebben a fejezetben példákon keresztül be fogjuk mutatni az D-Box nyelvi dobozok leírásának alapjait. Bemutatunk hibás példákat is, hogy felhívhassuk a figyelmet a bonyolultabb esetekre. A fejezetben szereplő példák megértése után egyszerű, és összetett esetek leírására is képesek leszünk. Megismerjük a csatornaindítási problémákat, valamint a dinamikus doboz indítási működést is.

A *D-Box* nyelv egy koordinációs nyelv, mely elsősorban a számítást végző csomópontok közötti kommunikációt írja le, s ily módon az elosztott funkcionális számítást támogatja.

A *D-Box* szó a Distributed Box rövidítéséből keletkezett. Kiindulási alapnak vettük a kommunikációs minta grafikus megjelenítését, ahol a számítási csomópontokat egy kis négyzet (Box) jelöli. A csomópontok között húzódó vonalak az adatáramlást megvalósító csatornákat szimbolizálják.



1.1. ábra. Számítási gráf

Egy számítási doboz három fő komponenssel rendelkezik:

- bejövő csatornák,
- számítási feladat,
- kimenő csatornák.

A számítási feladat a doboz definíció legfontosabb eleme, mivel ez képviseli a doboz tényleges funkcióját. Ez valójában egy funkcionális nyelven megfogalmazott kifejezés,

1.2. ábra. Számítási doboz

vagy függvény, melynek a kiértékeléséhez szükséges adatok külső helyről, egy másik dobozból érkeznek. A függvény a paraméterei segítségével fogadja az adatokat, elvégzi a számítási műveleteket és az eredményt, mint visszatérési értéket produkálja. A visszatérési értéket a doboz továbbítja a kimenő csatornákon át más dobozok felé.

1.1. A D-Box nyelvi definíciók ismertetése

Az 1.3. példában bemutatunk egy egyszerű elosztott számítás leírását *D-Box* nyelven.

D-Box definíció

```
1   BOX BoxID_1000  // Start doboz
2   {    1,                              // algráf azon.
3        { ( (null) ), memory},          // input csat
4        { ( null ), generator ,([Int]) },  // kifejezés
5        { (([Int],1001)), split1}       // output csat
6   }
7
8   BOX BoxID_1001  // Feldolgozo doboz
9   {    1,
10       { (([Int],1001)), join1},
11       { ([Int]), (func) ,([Int]) },
12       { (([Int],1002)), split1}
13  }
14
15  BOX BoxID_1002  // Stop doboz
16  {    1,
17       { (([Int],1002)), join1},
18       { ([Int],*World), WriteResult ,(*World) },
19       { ( null ), memory}
20  }
```

1.3. példa. Egy egyszerű példa D-Box nyelven

Az 1.3. példában szereplő dobozoknak a projekten belül egyedi azonosítóik vannak (rendre *BoxID_1000*, *BoxID_1001*, *BoxID_1002*). A dobozok mindegyike ugyanabba a projekt-alcsoportba vannak sorolva, ezt jelképezi a dobozok belsejében szereplő *1* algráf azonosító.

Az első (*BoxID_1000*) doboz generátor szerepet tölt be. Nincs input csatornája (a csatorna lista helyett *null* szerepel). A feldolgozó függvényének nincs input paramétere, de előállít egész számok egy listáját. Ennek a listának az elemei egy kimenő csatorna (azonosítója *1001*) felé kerülnek átadásra, amely az elemeket magába fogadja.

A második (*BoxID_1001*) doboz ezt a csatornát használja input forrásként, vagyis a csatorna olvasásakor a generátor függvény által előállított elemeket fogja feldolgozni. A feldolgozó függvény (*func*) az egész számok listájából egy újabb egész számok listáját állítja elő, melyet egy második csatorna (azonosítója *1002*) vesz át.

A harmadik (*BoxID_1002*) doboz ezen azonosítójú csatornát használja input forrásként, és a beérkező egész számokat egy *WriteResult* függvénynek adja át. Ezen függvény még egy **World* típusú paramétert is igényel, mely paramétert az I/O műveleteket végző függvények számára szükséges a *Clean* programozási nyelven. A függvény a beérkező adatokat fájlba menti, és az I/O műveletek végén előállt, megváltozott **World* értéket adja meg kimenetként. Ez az érték nem szállítható csatornán keresztül (helyreállítható típus), így ezen doboznak már nincs kimenő csatornája, ezt jelöli a *null* a csatorna-definíciós listán.

Mint látható, a doboz a függvénye részére a bejövő adatokat csatornákon keresztül nyeri ki. Ezekből kell a függvény paramétereit megkonstruálni. Ehhez valamilyen transzformációkat kell alkalmazni, melyet a doboz automatikusan végez. Egy ilyen csatorna intelligens, típusos adatfolyamnak is felfogható, melynek kezelése a Sor (*Queue*) összetett adatszerkezet műveleteihez hasonló műveletek segítségével történik.

Valójában a dobozok, és a közöttük húzódó csatorna a klasszikus *termelő-fogyasztó* problémában megjelenő két alkotóelem, és a közöttük feszülő *puffer* mintájára készült. Ugyanakkor, mint látni fogjuk, ebben a környezetben csak egy termelő és csak egy fogyasztó használhatja ezt a puffert.

1.2. A csatornák használata

A csatornákat egyedi sorszámok azonosítják[1]. Egy csatornára jellemzően két dobozban is szokás hivatkozni, az egyik (*termelő*) doboz számra ezen csatorna output, a másik (*fogyasztó*) doboz számára pedig input funkciót tölt be.

Elvi lehetőség van arra, hogy egyetlen csatornát több doboz is írjon, de ekkor a csatornába kerülő adatok sorrendje nemdeterminisztikus lesz, vagyis erősen attól függ,

[1] később látunk majd más azonosítási rendszert is

hogy az adott futási esetben a dobozok egymáshoz viszonyítva milyen (esetleg eltérő) sorrendben állították elő az adatelemeket, és helyezték el a csatornára.

Mivel a kiolvasás sorrendje garantáltan egyezik a beírás sorrendjével, ezen nemdeterminisztikus írási viselkedés végső soron nemdeterminisztikus kiolvasási viselkedést produkál, amely a teljes feldolgozási gráf nemdeterminisztikus viselkedéséhez vezetne. Ezen megfontolásból a *D-Box* nyelven több doboz nem használhatja ugyanazon csatornát output csatornaként.

Hasonló gondolatmenettel indokolható, hogy amennyiben több *fogyasztó* doboz is használná ugyanazt a csatornát inputként, úgy az végső soron a teljes gráf nemdeterminisztikus viselkedéséhez vezethetne, amely szintén nem megengedett.

Amennyiben egy csatorna csak egy doboz input csatornájaként viselkedne, de nem tartozna egyetlen dobozhoz sem, mint output csatorna, akkor egy olyan adatfolyamot kapnánk, amelyet nem táplálna semmi. Ekkor a *fogyasztó* doboz első olvasási művelete blokkolódna, végtelen idejű várakozást okozna, és rajta keresztül akár a teljes számítási gráf blokkolódásához vezethetne.

Hasonló helyzet állhatna elő, ha egy olyan csatorna keletkezne, amelyet egy doboz írna, de nem lenne doboz, amely olvasná a tartalmát. Minden csatorna rendelkezik azzal a képességgel, hogy valahány (véges mennyiségű) adatelemet képes átmenetileg tárolni (pufferelés). Amennyiben a csatornára író doboz ezt a mennyiséget túllépné, a szinkron írási művelet blokkolódna. Ez végső soron a doboz, és rajta keresztül a teljes számítási gráf blokkolódásához vezetne.

A fenti problémák elkerülése végett a csatornák pontosan egy dobozhoz tartoznak, mint input, és pontosan egy dobozhoz tartoznak, mint output csatorna.

Egyetlen kérdés maradt: szerepelhet-e egy csatorna ugyanazon doboz input és output csatornájaként is? Mivel a *D-Box* nyelv alapvetően nem tiltja a hurok kialakítását a számítási hálóban, így nem tiltja az egy elemű hurok kialakíthatóságát sem. Ennek megfelelően van arra lehetőség, hogy valamely csatorna, mint input és mint output szerepkört is betöltsön ugyanazon dobozban. Természetesen ilyenkor a doboz feldolgozó függvényét úgy kell megalkotni, hogy első lépésben ezen csatornát ne olvasni, hanem írni próbálja. A Clean lusta kiértékelési rendszerét kell ehhez felhasználni.

A csatorna által kezelt adatok típusa egy aránylag szűk körből kerülhet ki. Az alábbi típusok szerepelhetnek a szállítandó adatok alaptípusaként:

- *Int* 32 bites egész szám,
- *Real* 8 byte-os valós szám (double),
- *Char* 1 byte-on ábrázolt, ASCII kódtábla szerinti karakter,
- *Bool* 1 byte-on ábrázolt logikai érték (0=hamis, minden más igaz).

A fenti alaptípusokból észrevehetően hiányzik a *string*, de a számítási műveletek során ritkán van szükség ilyen típusú adatok küldésére és fogadására.

1.1. megjegyzés. A D-Box nyelven nincsenek operátorok definiálva, ezért a klasszikus értelemben véve nem történik műveletvégzés, sőt, maguk az értékek sem jelentkeznek kézzelfogható alakban. Nincsenek a fenti típusokba tartozó konstansok, literálok, nem alkothatók kifejezések. Az előzőekben megadott típusok gyakorlatilag minden programozási nyelven jelen lévő típusok, és bármilyen köztes réteg alkalmazása esetén megoldható a szerializációjuk.

Az alaptípusokra két típuskonstruktort lehet alkalmazni:

- *rekord*: rekord típus képzése. Egy rekordnak tetszőleges számú mezője lehet, melyeknek típusai a fent felsorolt alaptípusok, vagy rekordok lehetnek.
- *lista*: a lista egy tetszőleges (akár nulla) elemszámú sorozat képzése. A lista elemei elemi adattípusok, rekordok, vagy azokból képzett lista lehet (a lista maximum kétdimenziós lehet).

A rekord képzésére megszorítás, hogy a rekord elemei nem lehetnek lista típusúak. A rekord típuskonstruktor rekurzívan alkalmazható, a rekord mezői lehetnek rekord típusúak is.

A lista típuskonstruktor jele a *[ ]*, szögletes zárójelpár, ahol *[T]* esetén *T* jelöli a típust. A lista elemei lehetnek rekordok és listák is. Vagyis lehet listák listáját képezni, de csak ezen mélységig, nem lehet tehát listák listájának listáját képezni. *Ezen korlát a futtató rendszer protokolljának egyszerű fejlesztésével könnyedén megemelhető, felkészítvén nagyobb dimenziókra is. A jelenlegi D-Box környezet ezzel a korláttal működik.*

A rekordok, mint típus deklarálására a *D-Box* nyelvben a doboz-definíciókon kívül van lehetőség, mivel a típusdeklaráció nem képezi részét a doboz-definícióknak. Ugyanakkor látni fogjuk, hogy ez inkább csak adminisztrációs jellegű lépés, mivel a rekord típus deklarálásának önnmagában nincs haszna a D-Box nyelvben, mivel kifejezések, műveletek nem írhatók le. Az adott rekordtípust a Clean nyelvi felhasználó által definiált függvények fogják majd használni, ezért, az ezen függvényeket leíró Clean forráskódban is szerepeltetni kell a rekord definícióját (ahol a mezőknek azonosítót is kell adni).

Az 1.4. példában szereplő *sajatRekord* típus egy olyan rekordot ír le, mely négy mezőből áll. Mint látható, a mezők típusai fel vannak tüntetve, de nincs a mezőknek azonosító (név) adva. Ennek oka, hogy a *D-Box* szinten a mezők nevei nem fontosak, mert ezen a nyelven nincs szükség a mezőkre történő hivatkozásokra. A rekord típus neve (*sajatRekord*) is csak egy későbbi csatornadefinícióban kerül majd felhasználásra, mint a csatornán átáramló adatok alaptípusának megnevezése. A csatorna sem foglalkozik a rekord mezőinek azonosítóival, mivel mint látni fogjuk, a rekord mezőit elkülönülten nem kezeli.

D-Box típusdefiníció

```
typedef sajatRekord
 {
   Int,
   Real,
   Real,
   Bool
 }
```

1.4. példa. Rekord típus definíció *D-Box* nyelven

A rekord fogalma ezért átmenetet képez a hagyományos értelemben vett rekord fogalom (ahol a mezőknek is van azonosítójuk), és a funkcionális programozási nyelvekben értelmezett *tuple* fogalma között (ahol magának a konkrét *tuple* típusnak nincs azonosítója).

Egyszerű esetben egy csatornát két információ jellemez:

- az elemek típusa,
- a csatorna egyedi azonosítója.

Ennek megfelelően az alábbi példák helyes csatornadefiníciók:

- `(Int,1)`
- `([Real],2)`
- `([sajatRekord],3)`
- `([[Real]],4)`

Az első példa egy olyan csatornát ír le, amelynek azonosítója `1`, és egyetlen egy darab elemet fog a projekt során szállítani, egy egész számot. További elemeket nem fogad el tárolásra, és a következő elem olvasási kísérlete esetén *EoC* jelet (End of Channel, csatornavégjel) ad válaszul.

A `2`-es azonosítójú csatorna Real elemek listáját képes egyik dobozból a másikig továbbítani. Ezen elemek száma nullától a végtelenig terjedhet, mivel a csatorna képes kezelni végtelen elemszámú listát is.

A `3`-as azonosítójú csatorna gyakorlatilag mindenben megegyezik az előzőekben leírtakkal, azzal a különbséggel, hogy a csatorna elemeinek alaptípusa nem *Real*, hanem a korábben leírt rekordképpel megegyező. Ez általában is igaz: a tény, hogy a csatorna alaptípusa nem elemi alaptípusú, hanem rekord típusú, a csatorna működését nem változtatja meg. A rekord olvasása és írása a csatorna szempontjából ugyanúgy egyetlen lépés, mint elemi adattípusok esetén. Nincs lehetőség a rekordok egyetlen, vagy néhány mezőjének értékét a csatornára helyezi, sem a rekord egyetlen, vagy néhány mezőjének értékét kiolvasni.

A `4`-es azonosítójú csatorna olyan listát szállít, amelynek minden eleme egy-egy

lista, az adatelemek mindegyike egy-egy *Real* típusú érték. Az író oldalon az elemek feltöltése az első allista feltöltésével indul, majd allista vége jelet (*EoS*, *End of Sublist*) kell küldeni, és következhet a következő elemsorozat küldése, lezárva ezt is egy allista vége jellel. Az utolsó allista vége jel után egy lista vége jelet is kell küldenie (*EoL*, *End of List*). A fogadó oldal ugyanebben a sorrendben fogadja az adatokat, és az allista, lista végjeleket. Abban az esetben, ha valamely allista nem véges elemszámú, a következő allista feltöltésére nem fog sor kerülni a projekt futása alatt. Ez a fogadó oldalon is igaz - amíg valamely allista összes adata nem kerül kiolvasásra, addig a következő allista fogadása nem kezdhető meg.

Amennyiben két doboz egymással egy ilyen módon definiált csatornán keresztül kíván adatokat adni/kapni, úgy a futás során az alábbi lépések végrehajtása szükséges:[2]

- a doboz-példányok (futtatható alkalmazás) lekérik a csatorna pontos helyét (jellemzően IP cím és port) a futtató rendszertől,
- a futtató rendszer érzékeli, hogy az adott azonosítójú csatorna még nem fut, ezért elindítja a megfelelő típusú csatorna egy példányát utasítva, hogy vegye fel ezt az azonosítót.
- a csatornapéldány elindul, és helyét visszajelzi a futtató rendszernek,
- a futtató rendszer a helyet továbbítja a doboz példányok felé, akik e pillanattól kezdve önállóan (a futtató rendszertől függetlenül) kommunikálnak a csatornával.

Mivel ugyanazon csatornára jellemzően két doboz hivatkozik, valamelyik korábban éri el azt a pontot, hogy lekéri a csatorna helyzetét a futtató rendszertől. Ez bármelyik doboz lehet a kettő közül. Amennyiben közel egy időben keresik a csatornát, előadódhat az a helyzet, hogy mindkét kérés kiszolgálása miatt egy-egy csatorna kerül indításra. Ez a számítási gráf hibás működését eredményezné. Ezt a szituációt a futtató rendszer kezeli. Két megközelítés létezhet:

- a kéréseket sorba kell állítani, az első kérés beérkeztekor a futtató rendszer kezdeményezi a csatorna indulását, a második kérés beérkezésekor már ez folyamatban van, tehát a második kérés hatására nem történik meg az újbóli indítás.
- egyszerű döntésként csak az input célra használatos csatornák kérése okoz csatornaindítást. Amennyiben egy doboz output céllal kér egy csatornát – az mindig passzív várakozást eredményez, amíg az input kérés be nem fut, elindítván a csatornát.

Amennyiben a csatorna visszajelez, mindkét érintett doboz kiértesítésre kerül. Ha a csatorna adott idő (*timeout*) alatt nem jelentkezik be a helyzetével, az érintett dobozok értesítést kapnak a hibáról, és dönthetnek saját tevékenységük folytatásáról.

Amennyiben egy doboz nem rendelkezik input csatornával (pl. generátor doboz), úgy az input csatornák helyén a *null* jelölést kell használni. Hasonlóképpen, ha egy

[2] ezen működés később pontosításra kerül.

doboznak nincsenek kimenő csatornái, azok helyén is a *null* jelölést lehet használni. Az ilyen dobozok a számítási eredményeket továbbküldés helyett kiírhatják diszkre.

1.3. A feldolgozó kifejezés bemutatása

A doboz feldolgozó függvénye Clean nyelven megírt függvény vagy kifejezés. A *D-Box* nyelvnek nem feladata ezen függvény vagy kifejezés szintaktikai vagy szemantikai értelmezése, ezért az ide leírt „karaktersorozat" nem kerül további feldolgozásra. A *D-Box* fordító nem alkalmazható ezen kifejezés típusának elemzésére sem, ez nem feladata.

Ugyanakkor a kifejezés input paramétereit a *D-Box* nyelvi fordítónak ismernie kell, hogy a bejövő csatornákat rá tudja „kötni", mint a bemenő adatok forrását. Hasonlóképpen, a kifejezés eredményének típusát is ismernie kell, hogy azokat a kimenő csatornák felé továbbítani tudja.

Ezért a *D-Box* nyelvi doboz-definíció a kifejezés leírásán felül tartalmazza a kifejezés paramétereinek típusait, és az eredményének típusait. Mivel a doboz csatornáit össze kell tudni kapcsolni a kifejezés agumentumaival és visszatérési értékeivel, ezért ezeknek a típusa csak a csatornák által értelmezett típus lehet. (Mivel a csatorna adatot igen, kódot nem tud szállítani, nincs lehetőség magasabb rendű függvény eredményt adó kifejezést, sem ilyen típusú paraméterrel dolgozó függvényt a doboz kifejezéseként használni.)

Ha megtekintjük az 1.3. példa *DBox_1000* doboz definícióját, a feldolgozó kifejezés ott a *generator*, mely esetben a függvény *Clean* nyelvi deklarációja az alábbi formában írható fel: `generator :: [Int]`.

Ugyanezen példa *DBox_1001* doboz definíciójában szereplő feldolgozó kifejezés a *func*, melynek a *Clean* nyelvi prototípusa az alábbi leírású: `func::[Int]->[Int]`

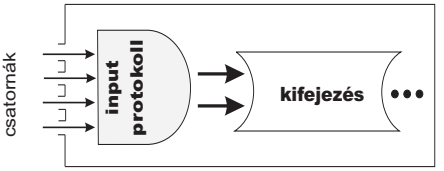
Ugyanezen példában szereplő *DBox_1002* dobozban lévő kifejezés a *WriteResult*, melynek a *Clean* nyelvi deklarációja az alábbi formájú:

```
WriteResult::[Int] *World -> *World.
```

A `*World`-ről jegyezzük meg, hogy az egy speciális típus, melyet a csatornák nem képesek szállítani, lévén hogy az csak az adott csomópontban értelmezett érték. Ez a külvilág (environment) állapotát írja le. Ezen érték nem szállítható, de szükséges azok a *Clean* függvények számára, amelyek I/O kezelést végeznek. Az ilyen típusú értéket az adott doboz-példány maga állítja elő, és kezeli. A D-Box nyelv számára a `*World` típus *helyreállítható* típusként van értelmezve.

1.4. Input protokoll ismertetése

Mint láttuk, a doboz input csatornái és a kifejezés paramétereinek típusa között léteznie kell összefüggésnek. A csatornák típusai és a kifejezés paramétereinek típusai egyszerű esetben megegyeznek. Ekkor minden egyes csatorna egy-egy paraméternek felel meg, a csatornák és a paraméterek közötti leképezés direkt módon történik.



1.5. ábra. Input protokoll

Bonyolultabb esetek is elképzelhetők, ezért szükség van egyfajta *mappingra*, amely a csatornák bejövő adatait leképezi a paraméterekre. A *D-Box* nyelven ezt *protokollnak* nevezzük. Jelenleg három ilyen protokoll létezik:

- *join1*,
- *joink*,
- *memory*.

1.4.1. A *join1* input protokoll

A *join1* protokoll a direkt csatorna → paraméter kötést írja le. Ekkor a csatornák száma és típusai rendre megegyeznek a kifejezés paramétereinek számával és típusaival (az esetleges helyreállítható típusú paramétereket leszámítva).

Az input protokollt az input csatornák felsorolása után kell megadni. Az 1.3. példában szereplő *BoxID_1001* doboz ezen részét kiemelve az 1.6. példán mutatjuk be újra.

D-Box definíció

```
1 BOX BoxID_1001  // Feldolgozo doboz
2 { 1,
3   {(([Int],1001)), join1},
4   { ([Int]), (func) ,([Int]) },
5   { (([Int],1002)), split1}
6 }
```

1.6. példa. Input csatornák és az input protokoll

Egy ilyen inputleírásban szereplő kifejezés típusa e ponttól adott, csakis az alábbi *Clean* nyelvi típus-definíciók közül lehet valamelyik:

- `func::[Int]->...`
- `func::[Int] *World->...`
- `func::*World [Int]->...`

A **World* paraméter létezése és pozíciója indifferens a működés szempontjából. A kifejezésnek valamely pozíción egy `[Int]` paraméterrel kell rendelkeznie, és egyéb paraméterei már csak helyreállítható típusúak lehetnek.

Amennyiben több input csatorna is lenne, akár eltérő típussal, az értelmezés akkor is ugyanaz marad. Az 1.7. példában a kifejezés típusa a példában megadottnak kell lennie (valamint szerepelhet a paraméterek között egy `*World` paraméter is tetszőleges pozíción).

D-Box definíció

```
1 BOX BoxID_1021
2   { 1,
3   {(([Int],1001),([[Real]],1002),([[Int]],1003)), join1},
4   { ([Int],[[Real]],[[Int]],*World), WriteResultX ,(*World) },
5   { ( null ), memory}
6 }
```

1.7. példa. Többcsatornás input

Figyeljük meg, hogy a kifejezés paramétereinek száma és típusa (a *World paramétert leszámítva) rendre meg kell egyezzen a bejövő csatornák számával és típusával.

Tuple paraméteres kifejezés

A *join1* protokoll segítségével szinte bármilyen további, bonyolultabb leképezés is megvalósítható, aránylag kevés programozási munkát indukálva. Amennyiben az 1.7. példában szereplő *WriteResultX* függvény valójában nem három paraméterrel rendelkezne, hanem egyetlen tuple-ben kívánná a három értéket fogadni, úgy egy egyszerű *wrapper* függvénnyel ez megoldható (1.8. példa).

A *wrapperTP* függvényt egyszerűen bele kell venni a doboz kifejezésbe, hogy a *join1* protokoll a csatornákat ezen wrapper függvénynek adhassa át, amely a paramétereket megfelelően transzformálva továbbadja a tuple paraméterezésű *WriteResultTP*-változatnak. Ez esetben az 1.7. doboz kifejezése az 1.9. példa szerintire módosul.

Clean nyelvi kód

```
1 wrapperTP::[Int] [[Real]] [[Int]] -> ([Int],[[Real]],[[Int]])
2 wrapperTP a b c = (a,b,c)
3
4 WriteResultTP:: ([Int],[[Real]],[[Int]]) *World -> *World
```

1.8. példa. Wrapper függvény *tuple*-re alakításra

D-Box definíció

```
1 BOX BoxID_1021
2 {  1,
3    { (([Int],1001),([[Real]],1002),([[Int]],1003)), join1},
4    {([Int],[[Real]],[[Int]]),*World), WriteResultTP wrapperTP ,(*World)},
5    { ( null ), memory}
6 }
```

1.9. példa. *Tuple*-re alakítás utáni doboz definíciója

Rekord paraméteres kifejezés

Hasonlóan egyszerű olyan wrapper függvényt írni, amely a csatornák adatait valamilyen rekord mezőibe helyezi el. Feltételezzük, hogy *Clean* nyelvi szinten létezik az 1.10 példában bemutatott felépítésű rekord.

Clean nyelvi kód

```
1 :: myRec {
2  a :: [Int],
3  b :: [[Real]],
4  c :: [[Int]] }
```

1.10. példa. *Clean* nyelvi rekord definíció

Elkészíthető az a transzformációs függvény, melynek paraméterezése illeszkedik a *join1* protokoll igényeire, és kimenete előálítja a rekord paramétert (lásd az 1.11. példa).

A feldolgozó kifejezésbe természetesen bele kell foglalni a *wrapper* függvényt, hogy a protokoll felé az elvárt paraméterezést mutathassa, és végrehajthassa a paraméter transzformációt. Ez esetben az 1.7. példában szereplő doboz kifejezése módosul (1.12. példa).

Clean nyelvi kód

```
1 wrapperWR_REC::[Int] [[Real]] [[Int]] -> myRec
2 wrapperWR_REC a b c = myRec(a,b,c)
3
4 WriteResult_REC:: myRec *World -> *World
```

1.11. példa. Wrapper függvény *rekord*ra alakításhoz

D-Box definíció

```
1 BOX BoxID_1021
2 { 1,
3  { (([Int],1001),([[Real]],1002),([[Int]],1003)), join1},
4  {([Int],[[Real]],[[Int]]),*World), WriteResult_REC wrapperWR_REC ,(*World)},
5  { ( null ), memory}
6 }
```

1.12. példa. *Rekord*ra alakítás utáni doboz definíciója

Lista-paraméteres kifejezés

Teljesen hasonló elvek alapján lehet olyan wrapper függvényt készíteni, amely a bejövő listák alapján egy magasabb szintű, listák listáját készíti el (1.13. példa).

Clean nyelvi kód

```
1 wrapperL::[Int] [Int] [Int] *World -> ([[Int]],*World)
2 wrapperL a b c = [a,b,c]
3
4 WriteResultL:: [[Int]] *World -> *World
```

1.13. példa. Wrapper függvény a *listá*ra alakításhoz

A listák listájává alakítás feltétele, hogy minden bejövő csatorna azonos típusú legyen, hiszen a lista homogén adatszerkezet. Ezért a korábbi példákhoz képest nem csak a wrapper függvényt, hanem az input csatornák típusait is módosítani kellett (1.14. példa).

Kombinált wrapper függvény

A fenti esetek (tuple, rekord, listák listájává alakítás) nem csak önállóan, de egymással együttműködve is alkalmazhatók, valamint a wrapper függvénynek van lehetősége a paraméterek sorrendjét is megváltoztatni (lásd az 1.15. példában bemutatott wrapper függvényt, mely illeszkedik az 1.16. példában szereplő doboz-definícióhoz).

D-Box definíció

```
1 BOX BoxID_1024
2 {    1,
3   { (([Int],1001),([Int],1002),([Int],1003)), join1},
4   { ([Int],[Int],[Int],*World), WriteResultL wrapperL ,(*World) },
5   { ( null ), memory}
6 }
```

1.14. példa. *Lista*ra alakítás után a doboz definíciója

Clean nyelvi kód

```
1 wrapperXX::[Int] [[Real]] [Int] *World -> ([[Real]],[[Int]],*World)
2 wrapperXX a b c  w = (b, [a,c], w)
3
4 WriteResultXX:: [[Real]] [[Int]] *World -> *World
```

1.15. példa. Kombinált wrapper függvény

Mint láthattuk, a fenti *wrapper* függvények egyszerűek, létrehozásuk nem okoz gondot a Clean nyelven, segítségükkel kellő rugalmasságot érhetünk el. Ezért a *join1* protokoll működése akár elegendő is lehetne. Azonban minden ilyen wrapper függvény azon az elven alapszik, hogy a csatornák száma (és típusa) ismert. Későbbiekben fogunk látni olyan esetet, amikor a típus ismert fordítási időben, de a csatornák száma nem.

1.4.2. A *joink* input protokoll

Az alábbiakban bemutatunk egy újabb protokollt, melynek szükségessége ezen a ponton nem látszik egyértelműen, hiszen egy megfelelően megírt *wrapper* függvény tudja helyettesíteni. Később azonban ez a protokoll változat még hasznos lesz.

Amennyiben a csatornák típusa megegyezik, lehetőségünk nyílik arra, hogy a bejövő elemeket listává fűzzük. Ez a működés megegyezik az 1.13. példában bemutatott listává alakító wrapper függvény működésével. Ez alkalommal azonban nincs szükség *wrapper* függvény írására, az összefűzést a *joink* protokoll használatával érjük el (1.17. példa).

Mint a példában is látszik, az input csatornák típusa `[Int]`, amelyből a *joink* protokoll egy listák listáját készít (`[[Int]]`). Az egy dimenzióval mélyebb lista egyben a kifejezés argumentumának típusa is. Ekkor az eredeti kifejezést nem kell wrapper függvénnyel kompozícióba helyezni – a *joink* végzi el a wrapper függvény dolgát.

Amennyiben a *joink* protokollt kívánjuk használni, minden input csatornának egyforma típusúnak kell lennie, hiszen a *Clean* nyelven a lista homogén adatszerkezet, tehát csak ebben az esetben képezhető a listák listája. A képzett lista elemszáma megegyezik

D-Box definíció

```
BOX BoxID_1024
{ 1,
   { (([Int],1001),([[Real]],1002),([Int],1003)), join1},
   { ([Int],[[Real]],[Int],*World), WriteResultXX wrapperXX ,(*World) },
   { ( null ), memory}
 }
```

1.16. példa. Kombinált wrapper függvényhez illeszkedő doboz-definíció

D-Box definíció

```
BOX BoxID_1024
{   1,
 { ( ([Int],1001), ([Int],1002)), joink },
 { ([[Int]],*World), WriteResult , (*World) },
 { ( null ), memory}
}
```

1.17. példa. *joink* protokoll

a csatornák számával.

1.4.3. A *memory* input protokoll

A *memory* input protokoll akkor használatos, amikor nincsenek bejövő csatornák, és emiatt csatorna → paraméter mappingra sincs szükség. Ez olyan dobozok esetén használható, amelyek generátor szerepet töltenek be a számítási gráfban.

D-Box definíció

```
BOX BoxID_1026
{  1,
 { ( (null) ), memory },
 { (*World), generate , ([Int],*World) },
 { ( [Int],1001 ), split1}
}
```

1.18. példa. *memory* protokoll

1.5. Output protokollok ismertetése

A kifejezés által előállított értékek általában csatornákra kerülnek, melyek az adatokat más dobozok felé továbbítják. Az input protokoll működéséhez hasonlóan itt is szükség van egy mechanizmusra, amely a kimeneti értékeket csatornákhoz köti.

A *D-Box* nyelv négyféle output protokollt ismer:

- *split1*,
- *splitk*,
- *splitf*,
- *memory*.

1.5.1. A *split1* output protokoll jellemzői

A legegyszerűbb esetben a kifejezés egyetlen visszatérési értékkel rendelkezik, melyet egyetlen csatornára kell rákötni.

Amennyiben a kifejezés eredményének típusa tartalmaz helyreállítható típusokat is, azokat a kimenő protokoll figyelmen kívül hagyja, mivel ezen típusú érték nem szállítható csatornán keresztül (1.19. példa). (A doboz definíció előtt megadtuk komment formájában a Clean nyelvi típusdefiníciót a *generate* függvényhez.)

D-Box definíció

```
1  // generate:: *World -> ([Int],*World)
2
3  BOX BoxID_1026
4  { 1,
5    { ( null ), memory },
6    { (*World), generate , ([Int],*World) },
7    { ( [Int],1001 ), split1}
8  }
```

1.19. példa. *split1* output protokoll

Hasonlóan egyszerű eset, ha az eredmény egy N tagú *tuple*, melynek elemeit más-más csatornákra kell rákötni. Egy ilyen esetben az sem okoz problémát, hogy a *tuple* tagjai eltérő típusúak (1.20. példa).

Természetesen itt is van arra lehetőség, hogy a kifejezés eredményét ne közvetlenül a protokoll kapja meg és dolgozza fel, hanem először egy *wrapper* függvény alakítsa át azt a kívánalmaknak megfelelően. Az 1.21. példa azt mutatja be, hogy amennyiben a kifejezés egy rekordot eredményezne, hogyan érhetjük el azt, hogy minden mezőjét más-más csatorna kapja meg továbbításra.

D-Box definíció

```
1 // generate:: *World -> ([Int],[[Real]],*World)
2
3 BOX BoxID_1027
4 { 1,
5   { ( (null) ), memory },
6   { (*World), generate , ([Int],[[Real]],*World) },
7   { ( ([Int],1001),([[Real]],1002)), split1}
8 }
```

1.20. példa. *split1* output protokoll **World* paraméterrel

Clean nyelvi kód

```
1 wrapperX::myRec2 *World -> ([Int],[[Real]],*World)
2 wrapperX a w = (a.mezo1, a.mezo2, w)
3
4 generate::*World -> (myRec2,*World)
```

1.21. példa. *wrapperX* függvény rekordból tuple-ra alakításhoz

A *wrapperX* függvényt kompozícióba kell helyezni a generátor függvénnyel, hogy a transzformációt el tudja végezni (1.22. példa).

D-Box definíció

```
1 BOX BoxID_1028
2 { 1,
3   { ( (null) ), memory },
4   { (*World), wrapperX generate, ([Int],[[Real]],*World) },
5   { ( ([Int],1001),([[Real]],1002)), split1}
6 }
```

1.22. példa. A *wrapperX* függvény használata

Az 1.23. példában azt mutatjuk be, hogy lehet három allistából álló listák listáját három különböző csatornára rákötni egy egyszerű *wrapper* függvény segítségével.

Az 1.24. példában látható a wrapper függvény használata a doboz-definícióban, kompozícióba kapcsolva a generátor függvénnyel.

Clean nyelvi kód

```
1  generate::*World -> ([[Int]],*World)
2
3  wrapperZ::[[Int]] *World -> ([Int],[Int],[Int],*World)
4  wrapperZ [a:b:c] w = (a,b,c,w)
```

1.23. példa. *wrapperZ* függvény listák kezelésére

D-Box definíció

```
1  BOX BoxID_1028
2  { 1,
3    { ( null ), memory },
4    { (*World), wrapperZ generate, ([Int],[Int],[Int],*World) },
5    { ( ([Int],1001),([Int],1002),([Int],1003)), split1}
6  }
```

1.24. példa. A *wrapperZbox* függvény használata

1.5.2. A *splitk* output protokoll ismertetése

A következőkben bemutatásra kerülő *splitk* protokoll használata szintén kiváltható a *spli1* protokoll használatával, valamint egy megfelelően megírt *wrapper* függvény segítségével. Az esetet bemutattuk az 1.23. példa kódban, amikor is a kiértékelő függvény ismert elemszámú listák listáját állítja elő, s az allistákat egy wrapper függvény különíti el, és a *split1* protokoll küldi tovább.

Ezt a feladatot látja el a *splitk* protokoll is, amely a generált listák allistáit elkülöníti, majd más-más csatornákon továbbítja. Alkalmazásánál megkötés, hogy a kifejezés más értéket nem is produkálhat (a helyreállítható típusú értékeken kívül), és a listának pontosan annyi elemszámú allistát vagy elemet kell tartalmaznia, mint a definiált output csatornák száma. Szintén megkötés, hogy az output csatornák típusa meg kell egyezzen az allisták típusával.

Amennyiben a kifejezés eredmény listája nem megfelelő számú allistát tartalmaz (az 1.25. példában kötelezően három allistát), úgy a *splitk* protokoll futási hibát eredményez. A *splitk* protokoll működése szerint az allistákat párhuzamosan tölti fel az output csatornákra. (Valójában, mint látni fogjuk, a *splitk* a *splitf* protokoll speciális esetének tekinthető.)

D-Box definíció

```
1 BOX BoxID_1029
2 { 1,
3   { ( null ), memory },
4   { (*World), generate, ([[Int]],*World) },
5   { ( ([Int],1001),([Int],1002),([Int],1003)), splitk}
6 }
```

1.25. példa. A *splitk* protokoll alkalmazása

1.5.3. A *splitf* output protokoll bemutatása

A *splitk* protokoll erős megkötései nem minden esetben teszik lehetővé az alkalmazását: a kifejezés által produkált allisták száma meg kell egyezzen az output csatornák számával. Ez jelentős programozói plusz munkát okozhat (vagy teljesítmény-csökkenést) egy olyan esetben, amikor a feldolgozó függvény a *master-slave* modell alapján a beérkező adatelemeket egyenlő hosszúságú allistákba kívánja darabolni, hogy a *worker* dobozok azokat feldolgozhassák. Amennyiben nem ismert előre a beérkező adatok mennyisége, a *master* doboz nem képes helyesen meghatározni a vágási pontokat. Ilyenkor a vágáshoz vagy megvárja, míg az összes input adat beérkezik (meghatározván azok számát, a vágási pontokat), vagy valamilyen heurisztikával megsejti a mennyiséget, kockáztatván a hibás döntést.

A *splitf* protokoll a kifejezés által előállított listák listáját elemeire szedi, és az allistákat a kimenő csatornákra helyezi. Azonban nem követeli meg, hogy az allisták száma megegyezzen a csatornák számával, sőt, általában az allisták száma jelentősen meghaladja a csatornák számát. A *splitf* protokoll sorban dolgozza fel az allistákat, minden egyes allista számára választván egy kimenő csatornát. A választás során nem ciklikusan veszi sorra a következő csatornát, hanem kiválasztja az első olyan csatornát, amelyik szabad (free) státuszú, vagyis a hozzá tartozó *worker* doboz már elég sok elemet kiolvasott a csatornáról. Ennek megfelelően, ha valamely feldolgozó doboz gyorsabb a társainál, akkor több munkát kap.

Az 1.26. példában szereplő doboz kódja valójában majdnem teljesen megegyezik az 1.25. példában szereplővel. Az output protokoll neve megváltozott. De egyéb változás is van, amely nem látszik a doboz kódjában: a *splitk* protokoll használata esetén a *generate* függvénynek a megadott három output csatorna miatt három elemű listát kell eredményeznie. A *splitf* esetén azonban ez nem követelmény.

D-Box definíció

```
1 BOX BoxID_1030
2 { 1,
3   { ( (null) ), memory },
4   { (*World), generate, ([[Int]],*World) },
5   { ( ([Int],1001),([Int],1002),([Int],1003)), splitf}
6 }
```

1.26. példa. A *splitf* protokoll alkalmazása

1.5.4. A *memory* output protokoll

A *memory* output protokoll akkor használatos, amikor nincsenek kimenő csatornák, és emiatt paraméter → csatorna kötésre sincs szükség. Ez olyan dobozok esetén használható, amelyek a feldolgozási lánc utolsó elemeiként a kapott adatokat diszkre mentik, vagy egyéb végfeldolgozási műveletet hajtanak végre rajtuk.

D-Box definíció

```
1 BOX BoxID_1002
2 {   1,
3   { (([Real],1002),([Real],1003)), joink},
4   { ([[Real]],*World), WriteResult ,(*World) },
5   { ( (null) ), memory}
6 }
```

1.27. példa. *memory* output protokoll

1.6. Az algráf azonosító szerepe

Egy számítási gráf elemei olyan dobozok, amelyek feldolgozó függvényeket zárnak magukba. A dobozok indulása a kommunikációs csatornák indulását is okozza, mivel a dobozok az általuk használt input vagy output csatornák helyzetét lekérve implicit módon kikényszerítik azok indulását. Ugyanakkor az egyik doboz indulása, vagy a hozzá tartozó csatornák indulása nem okozza további dobozok indítását. A futtató rendszer a projektet alkotó dobozok indításait egy időben végzi.

Egy projektben előfordulhat, hogy bizonyos dobozok indulása futás közbeni feltételtől függ. Ez a feltétel alapozható a feldolgozandó elemek számára, vagy egyéb tényezőre. Szükség van olyan mechanizmusra, ami a számítási gráf dobozainak egy halmazát dinamikusan, futás közben indítja el.

Egy projektet alkotó dobozokat az algráf azonosító (*sub-graph id*) alapján lehet részhalmazba sorolni. Minden részhalmaznak saját egyedi sorszáma van. Az ugyanazon részhalmazba sorolt dobozok mindegyike ugyanazt az azonosítót használja. A részhalmazokat nem kell külön deklarálni, a projektet alkotó dobozok feldolgozása során a fordító kigyűjti az előforduló azonosítókat, és ezek alapján dől el, hogy hány részhalmazból áll a projekt.

Az 1-es azonosító szerepe kitüntetett. Ezen részhalmazba sorolt dobozok indulnak el a projekt indulásakor. Nevezzük ezt a részhalmazt alkotó dobozokat *statikus dobozoknak*.

Amikor a futtató rendszer elindít egy algráfot, az indított dobozok számára egyedi *szálazonosítót* (*thread-id*) generál[3]. Ugyanazon projekten belüli ugyanazon algráf többször is elindítható, ezért futás közben a doboz azonosító nem egyedi, de a szál azonosító, és a doboz azonosító együtt már egyedi.

A dinamikusan indított dobozok egymással szintén csatornákon keresztül tudnak adatot cserélni. Ezen csatornák indítását a dinamikusan indított dobozok kényszerítik ki, csakúgy, mint a statikus dobozok esetén. Az algráf az őt indító doboztól kapja a feldolgozandó adatokat, és egy másik, az indító dobozzal egy szálba tartozó doboznak adja át a kiszámított adatokat. Így együttműködve összefüggő számítási hálót alkotnak.

D-Box definíció

```
1  BOX BoxID_1102
2  { [1],
3    { (([Real],1001),([Real],1002)), joink},
4    { ([[Real]]), func ,([Int]) },
5    { ( [Int],1003 ), split1}
6  }
7
8  BOX BoxID_1103
9  { [2],
10   { (([Real],1005)), join1},
11   { ([Real]), func ,([Real]) },
12   { ( [Real],1001 ), split1}
13 }
```

1.28. példa. Hibás felépítésű gráf

Az 1.28. példában szereplő *BoxID_ 1003* doboz a *2*-es algráfba tartozik, így indulása futás közbeni feltételtől függ. Amennyiben a futási körülmények miatt ezen doboz nem

[3] ez sok hasonlóságot mutat az operációs rendszerek témakörében szereplő *processz azonosító* fogalmával.

indul el, nem lesz olyan doboz, amely a *1001*-es csatornára elemeket helyezne el. Ez alapján a *BoxID_1102*-es doboz nem tudna adatokat olvasni az input csatornájából, ezen olvasási kísérlete blokkolást idézne elő. Ezen gondolatmenetből az következik, hogy amennyiben egy dinamikusan indított doboz valamely statikus doboz input csatornáját kívánja táplálni, úgy az futási problémát okozhat. Természetesen nem megoldás, hogy ugyanezen csatornára egy másik statikus doboz is hivatkozzon, hiszen a dinamikus doboz indítása esetén már többen használnák ezt a csatornát, ami nem megengedett.

Hasonló gondolatmenettel indokolható, hogy a dinamikus dobozok nem használhatják másik algráfbeli doboz ily módon definiált (rögzített azonosítójú) output csatornáját, mint adatforrást. Ugyanis ha a dinamikus doboz nem indulna el, a csatorna véges tárolási képessége miatt ez idővel a termelő doboz blokkolódásához vezethetne.

Ugyanakkor a dinamikusan indított dobozok elvileg használhatnak fix azonosítójú csatornákat, amennyiben mind a termelő, mind a fogyasztó doboz ugyanazon algráfba tartozik. Ugyanis a dinamikusan indított algráf minden egyes eleme automatikusan, egy időben indul el, ezért nem fordulhat elő, hogy csak a termelő, vagy csak a fogyasztó doboz indul el.

Ez a viselkedés mégsem megengedett, mivel ez esetben az algráf csak egy példányban indítható el egy projekt futásának időszaka alatt. Nyilvánvaló, ha a dobozok fix azonosítójú csatornákat használnak, a második példányban indított algráf dobozai ugyanazon csatornákat, az előző példány csatornáit használnák, amely nemdeterminisztikus viselkedéshez vezethetne.

Az előző gondolatmenetből következik: a dinamikusan indított (nem *1*-es azonosítójú algráfok) dobozai nem fix, hanem *automatikus kiosztású* azonosítókkal ellátott csatornákat használnak.

1.6.1. Az *AUTO* azonosítójú input csatorna

Amennyiben egy doboz input csatornája nem előre rögzített azonosítóval rendelkezik, hanem futás közben kell generálni az azonosítóját, úgy azt az *AUTO* kulcsszóval kell megjelölni:

A doboz indulásakor az alábbi események következnek be:

- a doboz kéri a futtató rendszertől egy adott típusú (a példában */Int/*) csatorna indítását,
- a futtató rendszer kezdeményezi a megfelelő típusú csatorna indulását, és várja a csatorna bejelentkezését,
- a csatorna elindul, mivel nem rendelkezik azonosítóval, kér egyet a futtató rendszertől,
- a csatorna a helyét (IP cím és port), valamint a futás közben kapott azonosítóját visszajelzi a futtató rendszernek,

D-Box definíció

```
1  BOX BoxID_1105
2  {    2,
3    { (([Real],AUTO)), joink},
4    { ([[Real]]), func ,([Int]) },
5    { ( [Int],AUTO), split1}
6  }
```

1.29. példa. *AUTO* azonosítójú input csatorna

- a futtató rendszer a kapott információkat továbbítja a csatorna indítását kérvényező doboz számára, aki az adatok birtokában a továbbiakban már önállóan tartja a kapcsolatot a csatornával.

1.6.2. Az *AUTO* azonosítójú csatorna output párja

Belátható, hogy az *AUTO* azonosító mint output csatorna azonosító nem szerepelhet párban egy másik *AUTO* azonosítójú input csatornával. Figyeljük meg az 1.30. példában szereplő két doboz-definíciót!

D-Box definíció

```
1  BOX BoxID_1102
2  {    2,
3    { (([Real],AUTO),([Real],AUTO)), joink},
4    { ([[Real]]), func ,([Int]) },
5    { ( [Int],AUTO), split1}
6  }
7
8  BOX BoxID_1103
9  {    2,
10   { (([Real],AUTO)), join1},
11   { ([Real]), func ,([Real]) },
12   { ( [Real],AUTO), split1}
13 }
```

1.30. példa. *Hibás AUTO* azonosító használat

Az 1.30. példában szereplő dobozok mindegyike a *2*-es azonosítójú algráfba tartozik, vagyis dinamikus indítású dobozok. A csatornák azonosítói futás közben generált értékek lesznek. Azonban a példában is látszik, hogy nincs kapcsolat a dobozok között:

nem felismerhető, hogy ugyanazon csatornára hivatkoznak. Az előzőekben ismertetett háttérbeli működés miatt a futtató rendszer minden egyes doboz minden egyes csatornáját elindítaná, és egyedi azonosítókat osztana ki. Ennek következtében a dobozok között nem lenne közös csatorna, nem működne az adatáramlás.

1.6.3. A *connBox* output csatornák

Amennyiben olyan dobozt készítünk, amely egy *AUTO* azonosítójú input csatornára kíván csatlakozni, speciálisan kell azt jelölni. Az alábbi azonosító adatokat kell megadni:

- melyik doboz csatornájára kívánunk csatlakozni (doboz-azonosító segítségével),
- a doboz melyik példányára kívánunk felcsatlakozni (szál-azonosító),
- a doboz hányadik csatornájától kezdjük a felcsatlakozást,
- a keresett csatornák milyen típusúak (a csatornatípust mindkét oldalon meg kell adni).

A keresett doboznak „auto” típuson felül más típusú csatornái is lehetnek. Ezeket át kell lépnünk, ki kell hagynunk. A csatlakozás során ezért meg kell adni egy kezdő sorszámot. Ha például a doboz harmadik csatornájára kívánunk felcsatlakozni, akkor az első két csatornáját kell átlépnünk.

A csatlakozást speciális kulcsszóval, *connBox*[4] kell megadni, paramétereiben felsorolva az előzőekben említett információkat:

D-Box definíció

```
1 BOX BoxID_1104
2 {    2,
3   { ([Real],AUTO), join1}
4   { ([Real]), func ,([Real]) },
5   { ( connBox thisThread BoxID_1102 0 ([Real]), split1},
6 }
```

1.31. példa. *connBox* csatlakozás egy *AUTO* input csatornára

A *thisThread* jelöli a csatlakozni kívánó doboz példány szálazonosítóját. A fenti eset azt jelöli, hogy a *BoxID_1102* doboz példányok közül azt a dobozt keressük, amely velünk egy algráfban, egy szálban került indításra. A *0* sorszám jelzi, hogy ezen doboz 0. sorszámú csatornájára kívánunk felcsatlakozni (a sorszámozás nullától indul). Ezen felül használható még az *ancestorThread* jelölés, amely az indító szál azonosítóját helyettesíti. A saját és szülő szálakon kívül további szálakra nem lehet hivatkozni.

A *connBox* csak output leírásban használható, mivel csakis input csatornák felé történő csatlakozásra használható.

[4] connBox = connect to a box

1.6.4. A *startGraph* alkalmazása al-gráfok indítására

Minden doboznak van lehetősége további dobozok dinamikus indítására. Ezzel a lehetőséggel elsősorban a fő szál dobozai élhetnek, de az általuk indított algráf dobozai is indíthatnak további algráfokat.

Az algráf egyik doboza kitüntetett szerepkörű (*belépő doboz*). Ezen doboz input csatornáit az algráfot indító doboz fogja táplálni adatokkal.

Az algráfban ezen felül általában van még egy speciális szerepkörű doboz (*végfeldolgozó doboz*). Ezen doboz kimenő csatornáit az indító szál valamely dobozának input csatornáira kell rákötni, hogy az algráf számítási eredményeit *visszajuttassa* a szülő algráf valamely doboza számára.

D-Box definíció

```
1  BOX BoxID_1001  // DDivideN
2  {    1,
3       { (([Int],1001)), join1},
4       { ([Int]), (s_divide 2) ,([[Int]]) },
5       { (startGraph 2 ‘lengthOf result‘ BoxID_1002 ( [Int] )), splitk}
6  }
7
8  BOX BoxID_1002  // DLinear
9  {    2,
10      { (([Int],auto)), join1},
11      { ([Int]), ((add 2)) ,([Int]) },
12      { (connBox ancestorThread BoxID_1004 ( [Int] )), split1}
13 }
14
15 BOX BoxID_1004  // DMerge
16 {    1,
17      { (autoConnBox thisThread BoxID_1001 ( [Int] )), joink},
18      { ([[Int]]), (mergeSort lessThan) ,([Int]) },
19      { (([Int],1005)), split1}
20 }
```

1.32. példa. Algráf kezelése

Az 1.32. példában a *BoxID_1001* doboz tölti be az algráf indító szerepet. Kimenő csatornái helyén szerepel a *startGraph* használata, mivel az indításon kívül a *BoxID_1002* doboz input csatornáira is rá fog csatlakozni. Az algráf indítási darabszámát egy futás közben kiértékelt kifejezés adja meg (*lengthOf result*). Ezen kifejezés

numerikus egész értékkel kell visszatérjen, és a működés miatt 0-nál nagyobb értéket kell adnia. A megadott *2*-es azonosítójú algráf dobozai ennyiszer (N darab) kerülnek indításra. Mivel mindegyik indítási szálban keletkezik *BoxID_1002*-es doboz, mindegyiknek egy-egy saját `[Int]` típusú input csatornája van, ezért ugyanennyi darab output csatornát is kell képeznie. Ezt a *startGraph* automatikusan megteszi a numerikus érték és a típuslista alapján.

Ekkor válik nyilvánvalóvá, miért is van szükség a *splitk* protokollra. Mivel a kimenő csatornák száma csak a kifejezés futás közbeni kiértékelésekor derül ki, ezért a *wrapper* függvények a *split1* protokollal együttműködve sem tudnák ezt az esetet kezelni.

1.6.5. Az *autoConnBox* input csatornák szerepe

Az 1.32. példában a *BoxID_1004*-es doboz fogadja az algráfok felől érkező eredményeket. Ezen doboz input csatoráinak indítása nem kevés problémával jár. A doboz ugyanis tudhatja, hogy az algráf *végfeldolgozó doboza*, a *BoxID_1002* kimenő csatornája `[Int]` típusú, de nem tudja, hány ilyen algráf került indításra, mivel az indítást nem ő, hanem a *BoxID_1001*-es doboz végezte.

Vegyük észre, hogy a *BoxID_1004* doboznak annyi kimenő csatornát kell indítani, ahány algráfot a *BoxID_1001* indított, ezért az *autoConnBox* paramétereként ezen dobozt (az algráf indító dobozt) meg kell adni.

Az indítandó csatornák típusa nem kell, hogy megegyezzen ezen algráf indító doboz csatornáinak típusával, mivel nem vele fognak direkt módon kommunikálni, hanem a dinamikusan indíttott algráfok végfeldolgozó dobozai fognak felkapcsolódni ide. Ebben a példában ezek a *BoxID_1002*-es dobozok, amiknek `[Int]` típusú kimenő csatornái vannak, tehát a*BoxID_1004*-es doboznak is ilyen típusú csatornákat kell indítania.

További megoldandó probléma, hogy a végfeldolgozó dobozok output csatornáinak más-más input csatornákra kell dinamikusan felkapcsolódnia. Ezt a problémát a futtató rendszer fogja kezelni. A megoldás szerint a *BoxID_1004*-es csatorna a kivárás elvét használja, így nem indítja el a csatornákat, megvárja, amíg a végfeldolgozó dobozok elindítják azokat, és ő csak lekéri ezek azonosítóit, így minden végfeldolgozó doboz más-más csatornán fog felkapcsolódni hozzá.

Felmerül a kérdés, hogy akkor miért kell ismernie ezen doboznak az indított algráfok számát. Működhetne az az elképzelés is, hogy a végfeldolgozó dobozok csatornaindításakor minden esetben kapnánk egy jelzést, hogy újabb bejövő csatornáink vannak, és azok adatait is fel kell dolgoznunk. Vegyük észre, hogy a doboznak ismernie kell, hogy hány ilyen bejövő csatornára kell számítani, mert el kell tudnia dönteni minden csatorna visszajelentkezett-e, és nem következett-e be valamely indított algráfban futási hiba, amely megakadályozza a kapcsolatfelvételt!

1.7. Összegzés

Ebben a fejezetben bemutattuk a *D-Box* nyelv legfontosabb elemeit, egy-egy példán keresztül. Megmutattuk, milyen input és output protokollok léteznek, melyik milyen körülmények között alkalmazható. Bemutattuk milyen módon adhatóak meg egyszerű és bonyolultabb esetekben az input és output célokra alkalmazható csatornák.

A *D-Box* nyelv első tervezésekor a *splitf* protokoll még nem került be a nyelvbe. A terheléstől függő számítási gráfot a dinamikus algráf indítási lépések, a *startGraph* és a *autoConnBox* lehetőségekkel oldottuk meg. A dinamikus algráf indítás azonban nehézkesnek bizonyult, mivel nem volt egyszerű futás közben eldönteni a bejövő adatok alapján hány algráfot kell indítani dinamikusan, valamint maguk az indítások a projekt futásában sebesség szempontjából törést okoztak. Egyébként sem jelentett feltétlenül előnyt a dinamikus viselkedés, hiszen általában ismert a futáskor hány darab számítógép (N) áll rendelkezésre. Adott N számítógépen elvileg lehet több mint N dobozt indítani, de ekkor ugyanazon számítógépen több számítási doboz is elindul. Ez egyáltalán nem jó irányba befolyásolta a futás sebességet.

Mivel a gépek száma a projekt indításakor ismert, indokoltnak tűnt a projekt futása során indított feldolgozó dobozok számát is ennek ismeretében tervezni. *Zsók Viktória* javaslatára dolgoztuk ki a *splitf* protokollt. Ennek során a feldolgozó dobozok száma fix mennyiségű lesz, de a működés mégis tud alkalmazkodni dinamikusan a feldolgozandó adatok mennyiségének változásához. További előny, hogy amennyiben nem minden számítógép egyforma teljesítményű, a *splitf* protokoll működéséből fakadóan előnyben fogja részesíteni a nagyobb teljesítményű feldolgozó dobozokat.

Ugyanakkor a korábbi megoldás, a *startGraph* és *autoConnBox* nem került eltávolításra a D-Box nyelvből. A beágyazott D-Clean kifejezések ([11].publikáció) támogatása miatt továbbra is részét képezik a nyelvnek. De egyéb esetekben inkább a *splitf* protokoll használatát javasoljuk.

2. fejezet

A D-Box nyelv statikus szemantikai leírása

A D-Box nyelv alapvető szintaktikai leírását EBNF formában, valamint *lex* és *yacc* nyelvi alakban a B. függelékben adjuk meg. Az alábbiakban kiegészítjük ezt a leírást a statikus szintaktikai szabályokkal, melyek megadják, mikor helyes (illeszthető) a doboz input csatornái és az alkalmazott protokoll mapping a kifejezés argumentumaira (számban, típusban, sorrendben), illetve a számítás eredménye az alkalmazott output protokoll szerint illeszthetőek-e a kimeneti csatornákra (számban, típusban, sorrendben). Ezen felül megadjuk, hogyan vizsgálható, hogy a projektet alkotó doboz definíciók összefüggő számítási hálót alkotnak-e (minden csatornát pontosan egy doboz ír és olvas, nincsenek felesleges csatornák).

2.1. Az átvihető típus fogalma

Először tisztázzuk, melyek azok a típusok, amelyek a csatornákon átvihetőek. Ezen típusok felismerése kulcsfontosságú, amikor a kifejezés paramétereit és az alkalmazott protokollt egyeztetjük. Amennyiben a kifejezésnek olyan típusú paraméterei is vannak, amelyek csatornákon keresztül nem szállíthatóak, akkor azokat ettől független módon kell tudnunk előállítani. Az ilyen típusokat **helyreállítható típusoknak** nevezzük, Clean nyelv esetén ilyen pl. a `*World` típus. Ha egy kifejezés leírásában használt típus nem is szállítható, nem is helyreállítható, az szintaktikai hibára utal.

2.1. definíció (Nyelvi alaptípus). A $T_{\tau a} = \{Int, Real, Char, Bool\}$ típusok halmazát *D-Box nyelvi alaptípusnak* nevezzük. Ezen felül τ_a-val jelöljük a $T_{\tau a}$ halmazt alkotó típusokba tartozó értékek halmazát ($\tau_a = \bigcup_{t \in T_{\tau a}} t$).

2.1. megjegyzés. A D-Box nyelvi szinten ezek azok a típusok, melyek a Clean és C nyelvi interfészen keresztül problémamentesen szerializálhatóak.

2.1. jelölés (Rekord konstruktor). A továbbiakban $R(t_0, t_1, \ldots, t_{n-1})$ módon jelöljük a rekord algebrai típus konstruktorát, ahol t_i a rekord mezőinek típusait képviseli.

2.2. jelölés (Lista konstruktor). A továbbiakban $L(t)$ módon jelöljük a lista algebrai típus konstruktorát, ahol t a lista elemeinek típusát képviseli.

2.2. definíció (Egyszerű átvihető típus). Legyen $R^0 \coloneqq T_{\tau a}$. Jelölje $i \geqq 1$ esetén R^i azoknak a típusoknak a halmazát, amelyek az R rekordkonstruktor segítségével képezhetőek valamely $R(t_1, t_2, \ldots, t_n)$ módon, ahol $t_1, t_2, \ldots, t_n \in \bigcup_{k=0}^{k<i} R^k$ a rekord mezőinek típusai. A $T_{\tau r} \coloneqq \bigcup_{k=0}^{\infty} R^k$ típusok halmazát *egyszerű átvihető adattípusnak* nevezzük. Ezen felül jelölje τ_r az ezen típushalmazt alkotó típusokba tartozó értékek halmazát.

2.2. megjegyzés. A $T_{\tau r}$ típushalmazba beletartoznak a nyelvi alaptípusok, valamint az olyan rekordtípusok, ahol egy-egy mező típusa akár maga is rekord típusú lehet (tetszőleges mélységben).

2.3. definíció (Átvihető típus). Valamely T_τ típust *átvihető adattípusnak* nevezünk, ha $T_\tau \in T_{\tau r}$, vagy $\exists t \in T_{\tau r}$, hogy $T_\tau = L(t)$ vagy $T_\tau = L(L(t))$ (ahol L a lista-konstruktor). Ezen felül jelölje τ a T_τ típushalmazt alkotó típusokba tartozó értékek halmazát.

2.3. megjegyzés. A T_τ típus definíciója szerint a lista-konstruktort csak egy vagy két dimenziós listák képzésére lehet használni. Ezen korlát csak a jelenlegi *D-Box* fordító és futtató rendszeri implementáció része. Mint látni fogjuk a protokollok és a szerializáció fejlesztésével a listák egymásba ágyazhatóságának mélysége növelhető.

2.4. megjegyzés. A T_τ típus a *D-Box* számítási csomópontok között húzódó csatornák által kezelt adattípus.

2.4. definíció (Kezelhető típus). Valamely τ' típust *kezelhető adattípusnak* nevezünk, ha a $\tau' \in T_\tau$, vagy $\exists t \in T_\tau$, hogy $\tau' = L(t)$. Jelölje $T_{\tau'}$ a kezelhető adattípusokból alkotott típushalmazt.

2.5. megjegyzés. A τ' típus legfeljebb egy dimenzióval lehet bővebb mint egy konkrét átvihető típus. Bevezetésének oka, hogy a *joink* protokoll a bejövő csatornákat egyesíti, egy dimenzióval bővíti. Legalább két darab [[Int]] input csatorna esetén a *joink* `[[[Int]]]` típusú eredményt állít elő. Ez az előállított típus még alkalmazható a kifejezés argumentumaként, de szállítani a csatornákon már nem lehetne (nem átvihető típus). Hasonlóan, ha egy kifejezés pl. `[[[Int]]]`-et állít elő, az általában nem vihető

át a csatornákon (nem átvihető típus), de a *splitk* protokoll erről leveszi a külső réteget (csökkenti a lista dimenzióját), és a kimenő csatornákon már csak [[Int]] fog kimenni, ami átvihető típus. Ugyanezt az `[[[Int]]]`-t már nem lehetne *splitf* protokollal kiküldeni, mert az nem csökkenti a dimenziószámot.

2.5. definíció (Helyreállítható típus). A $T_\delta = \{*World\}$ típusból álló halmazt *helyreállítható* típusok halmazának nevezzük. Ezen felül jelölje δ a T_δ típushalmazt alkotó típusokba tartozó értékek halmazát ($\delta = \bigcup_{t \in T_\delta} t$).

2.6. megjegyzés. A T_δ halmazba azok a típusok tartoznak, melyek a csatornákon nem szállíthatóak, de erre nincs is szükség, ugyanis a csatorna adatfolyamát olvasó oldalon a D-Box nyelvi működéstől független módon az értéket elő lehet állítani. Az előállítás után a protokoll wrapper függvény beilleszti a kifejezés argumentumlistájába az értéket a megfelelő helyre. A *Clean* esetében a `*World` az egyetlen ilyen típus, amellyel ez megtehető[1].

2.3. jelölés (felismert típusok). A $T_\omega \coloneqq T_\delta \cup T'_\tau$ halmazt *előforduló* típusok halmazának nevezzük. Ezen felül jelölje ω ezen típushalmazt alkotó típusokba tartozó értékek halmazát.

2.7. megjegyzés. A T_ω halmazba tartozó típusok fordulhatnak elő egy D-Box definícióban. A csatornákat leíró típusok csak a T_τ típushalmazba eshetnek, de a kifejezés paramétereit, vagy argumentumait jellemző típusok között előfordulhat T'_τ és T_δ típusú kezelhető vagy helyreállítható típusnév is.

2.2. Saját jelölések bemutatása

A dolgozat szövegezésében használt ismertebb, vagy saját jelölési formákat az alábbiakban foglaljuk össze.

2.4. jelölés ($\mathbb{N}, \mathbb{N}_+$). Jelölje $\mathbb{N} = [0,1,\dots]$ nemnegatív egész számok halmazát, $\mathbb{N}_+ = [1,2,\dots]$ a pozitív egész számok halmazát.

2.5. jelölés (Hatványhalmaz). Jelölje $\mathcal{P}(H)$ a H halmaz hatványhalmazát.

2.6. jelölés (sizeof). A továbbiakban $sizeof(a)$-val jelöljük azt a műveletet, amely az a komponens (halmaz, sorozat) elemszámát határozza meg. Amennyiben a egy véges sorozat, úgy $sizeof(a)$ a sorozat elemszámát jelöli. Hasonlóan, ha a egy véges halmaz, úgy $sizeof(a)$ jelöli a halmaz elemszámát. Ha a egy végtelen elemszámú sorozat vagy halmaz, úgy $sizeof(a) = \infty$.

[1] Ez esetben nem a másik dobozbeli *World értéket állítjuk helyre, hanem egy helyi *World értéket, mely a fogadó oldali környezet állapotát képviseli, nem a küldőét.

2.6. definíció. Legyen $depth\colon \tau' \to \mathbb{N}$, $depth(x) = n$ függvény, ahol

- ha $\exists d \in T_{\tau r}$, hogy $x \in d$, akkor $n \coloneqq 0$,
- ha $\exists d \in T_{\tau r}$, hogy $x \in L(d)$, akkor $n \coloneqq 1$,
- ha $\exists d \in T_{\tau r}$, hogy $x \in L(L(d))$, akkor $n \coloneqq 2$,
- ha $\exists d \in T_{\tau r}$, hogy $x \in L(L(L(d)))$, akkor $n \coloneqq 3$,
- egyébként $n \coloneqq -1$.

2.8. megjegyzés. A *depth* függvény megadja, hogy egy adott adat hány dimenziós listaként értelmezhető. Ha x egy „elemi" típusú érték, a dimenzió-mélység 0. Amennyiben az x típusa egy egyszerű átvihető típusból az L lista-konstruktor segítségével egy lépésben előállítható, úgy a dimenzió-mélysége 1, stb.

2.7. definíció. Legyen $depth_T\colon T_{\tau'} \to \mathbb{N}$, $depth(t) = n$ függvény, ahol

- ha $t \in R^0$, akkor $n \coloneqq 0$,
- ha $t \in R^1$, akkor $n \coloneqq 1$,
- ha $t \in R^2$, akkor $n \coloneqq 2$,
- ha $t \in R^3$, akkor $n \coloneqq 3$,
- egyébként $n \coloneqq -1$.

(Az R^0, R^1, R^2, R^3 típushalmazok a 2.2. definícióban kerültek bemutatásra).

2.9. megjegyzés. A $depth_T$ függvény hasonló szerepet tölt be, mint a *depth*, de ezen változat paramétereként egy típust kap, és meghatározza, hogy hány dimenziós listaként értelmezhető ezen típus.

2.7. jelölés (Sorozat). A továbbiakban $\langle X \rangle$ módon jelöljük az X típusú elemekből álló sorozatok halmazát. A sorozatok elemeit 0-tól számozzuk. Amennyiben $D \in \langle X \rangle$ egy véges sorozat, úgy $D = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ jelölést használunk a konkrét elemek jelölésére. (Ekkor nyilvánvalóan $sizeof(D) = n$.) Amennyiben $C \in \langle X \rangle$ egy végtelen sorozat, úgy $C = \langle x_0, x_1, \ldots \rangle$ jelölést használjuk. Az üres (nulla elemű) sorozatot az $\emptyset$ szimbólummal jelöljük.

2.8. definíció (összefűzés). Amennyiben $A, B \in \langle X \rangle$ véges sorozatok, úgy értelmezzük a $\oplus\colon \langle X \rangle \times \langle X \rangle \to \langle X \rangle$, $\oplus(A, B) = C$ műveletet, ahol ha

$$A = \langle a_0, a_1, \ldots, a_{n-1} \rangle, B = \langle b_0, b_1, \ldots, b_{k-1} \rangle,$$

akkor

$$C \coloneqq \langle a_0, a_1, \ldots, a_{n-1}, b_0, b_1, \ldots, b_{k-1} \rangle.$$

Az $\oplus(A, B) = C$ felírást operátor alakban fogjuk használni $C \coloneqq A \oplus B$ módon.

2.9. definíció (Összefűzés). Amennyiben $A \in \langle X \rangle$ egy sorozat, és $b \in X$ egy elem, úgy kiterjesztjük a $\oplus$ operátor értelmezését az alábbiak szerint: $C \coloneqq A \oplus b$, ahol ha

$$A = \langle a_0, a_1, \ldots, a_{n-1} \rangle,$$

akkor

$$C \coloneqq \langle a_0, a_1, \ldots, a_{n-1}, b \rangle.$$

2.10. megjegyzés. Az $\oplus$ művelet segítségével nem csak két sorozatot lehet összefűzni, hanem egy sorozat végére elhelyezhetünk egy új elemet.

2.10. definíció (eltávolítás). Amennyiben $A \in \langle X \rangle$ véges sorozat, és $b \in X$ egy elem, úgy értelmezzük az $\ominus$ műveletet az alábbiak szerint: $C \coloneqq A \ominus b$, ahol ha $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$, akkor $H \coloneqq \{j : j \in [0, n-1], a_j = b\}$ indexhalmaz esetén

- ha $H = \emptyset$, akkor $C \coloneqq A$,
- különben $i \coloneqq min(H)$ index esetén $C \coloneqq \langle a_0, a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_{n-1} \rangle$.

2.11. megjegyzés. Az $\ominus$ művelet a sorozatból eltávolítja az adott elemet. Amennyiben az elem többször is szerepelne a sorozat elemei között, úgy a legkisebb sorszámú elemet távolítja el. Amennyiben az elem nem szerepel a sorozatban, úgy a sorozatot érintetlenül hagyja.

2.11. definíció (dekompozíció). Legyen $\hookleftarrow : \langle X \rangle \to X \times \langle X \rangle$, $\hookleftarrow (A) = (a', A')$ függvény, ahol ha $A\langle a_0, a_1, \ldots, a_{n-1} \rangle$ véges sorozat, akkor $a' \coloneqq a_0$ a sorozat első eleme, $A' \coloneqq \langle a_1, a_2, \ldots, a_{n-1} \rangle$ a maradék elemek. A továbbiakban ezt a függvényt operátor alakban $(a', A') \hookleftarrow A$ módon fogjuk jelölni.

2.12. megjegyzés. A $\hookleftarrow$ művelet (dekompozíció) leválasztja a sorozat első elemét, továbbá képzi azt a sorozatot, amelyből ezen első elem hiányzik. Ha a sorozat üres, akkor a művelet nincs értelmezve.

2.8. jelölés (boolean). A továbbiakban $\mathbb{B}$-vel jelöljük a $\{true, false\}$ halmazt (logikai értékek halmaza).

2.3. Előkészületek a statikus szemantika leírásához

Megadunk két olyan segédfüggvényt, amelyekre a későbbiekben hivatkozni kívánunk. A típusnevek értelmezése során típusok sorozatából kiemeljük a helyreállítható típusokat. A megmaradt típusokat a protokolloknak értelmezniük kell, és a csatornákkal össze kell tudni illeszteni. Ellenkező esetben szintaktikai hibát észlelhetünk.

2.12. definíció (NULL KIHAGYÁS). Legyen $*\colon \langle T'_\tau \rangle \times T_\omega \to \langle T'_\tau \rangle$, $*(S, x) = S'$ függvény, ahol ha $x \in T_\delta$ akkor $S' \coloneqq S$, különben $S' \coloneqq S \oplus x$. A $*$ függvényt mint operátor fogjuk jelölni $S' \coloneqq S * x$ módon. Kiterjesztjük ezen felül a $*$ operátort oly módon, hogy $S' \coloneqq a * b$ jelölje a $S' \coloneqq \emptyset * a * b$ működést $(a, b \in T_\omega)$.

2.13. megjegyzés. A $*$ operátor a kezelhető típusok listájához fűzi az x elemet, ha az is egy kezelhető típus. Amennyiben x egy helyreállítható típus, akkor nem fűzi a lista végére. Értelmezve van ezen felül az operátor nem csak sorozat és elem, hanem elem és elem típusú argumentumra is.

2.13. definíció (TÍPUSNEVEK ÉRTELMEZÉSE). Az $f_{styp}\colon \langle T_\omega \rangle \to \langle \tau' \rangle$, $f_{styp}(ts) = = ts'$ legyen olyan függvény, ahol ha $ts = \langle t_0, t_1, \ldots, t_{n-1} \rangle$ típusok sorozata, akkor $ts' \coloneqq t_0 * t_1 * \cdots * t_{n-1}$.

2.14. megjegyzés. Az f_{styp} függvény az előforduló típusok halmazából kiszűri a kezelhető típusokat oly módon, hogy a listából eltávolítja a helyreállítható típusokat. Ezen függvény a kifejezés argumentumainak, valamint eredményének típuslistáját fogja feldolgozni, eredményét statikus szemantikai helyesség szempontjából kell majd elemezni.

2.4. D-Box nyelvi definíciók felépítése

A D-Box nyelv EBNF formájú szintaktikai leírását a B. függelékben adtuk meg. Ebben a fejezetben a definíciót felépítő elemeket részleteiben tárgyaljuk, elsősorban a csatornaleíró struktúrákra koncentrálva. Ezen leírókra a statikus helyesség vizsgálatánál több ponton is hivatkozni fogunk. A csatorna leírók alapján az határozható meg, hogy a dobozokhoz hány darab, és milyen típusú csatorna tartozik. Ennek a listának először a hozzárendelt protokollhoz kell illeszthetőnek lennie. A dobozhoz tartozó csatornák későbbi vizsgálatokban is érdekesek lesznek: amikor azt fogjuk vizsgálni, hogy minden csatornához pontosan egy olvasó és író doboz tartozik-e.

Az input protokoll és a doboz input csatornái szoros egységet alkotnak. A *joink* esetén legalább egy csatornának kell léteznie, és minden csatorna típusa egyező kell legyen. A *join1* esetén nem kell minden csatornának egyező típusúnak lennie. A *memory* esetén pedig nem lehet jelen csatorna. Hasonló szabályok alkothatók az output protokollok esetére is, ahol a *split1* esetén különböző típusú csatornákat lehet használni, *splitk* és *splitf* esetén csak egyező típusúakat.

2.14. definíció (D-BOX NYELVI DEFINÍCIÓ). *D-Box definíciónak* nevezzük a

$$D_{box} = (BoxID, subGraphID, InpProt, ExpressionDef, OutProt)$$

formális ötöst, ahol a komponensek jelentése az alábbi:

- $BoxID \in String$ egyedi doboz-azonosító,
- $subGraphID \in \mathbb{N}$ egy egész szám, az algráf azonosító,
- $InpProt = (IChannels, IProtocoll)$ a doboz input igényét megfogalmazó csatornalista, és az input protokoll,
- $ExpressionDef = (inpTypes, expression, outTypes)$ a számítási kifejezés argumentumainak típusai, maga a kifejezés, és az eredmény típusainak felsorolása,
- $ExpressionDef.inpTypes \in \langle T_\omega \rangle$, az argumentumok típusainak felsorolása,
- $ExpressionDef.outTypes \in \langle T_\omega \rangle$, a számítási eredmény típusainak felsorolása,
- $OutProt = (OChannels, OProtocoll)$ a doboz output igényét megfogalmazó csatornalista, és az output protokoll.

2.15. megjegyzés. A további felépítő részek (InpProt.IChannels, InpProt.IProtocoll, OutProt.OChannels, OutProt.OProtocoll) részek felépítése a következő fejezetekben kerül tárgyalásra.

2.9. jelölés (Szálazonosítók). A $\mathbb{H} = \{ancestorThread, thisThread\}$ halmazt *szálazonosítók* halmazának nevezzük.

2.15. definíció (Input csatorna leírók). A B. függelék szerinti EBNF leírás alapján definiáljuk az input csatorna leírókat:

- $D_{null} = (null)$,
- $D_{fix} = (type, id)$, ahol $type \in T_\tau$ a csatorna típusa, $id \in \mathbb{N}_+$ a csatorna azonosítója,
- $D_{auto} = (type, auto)$, ahol $type \in T_\tau$ a csatorna típusa,
- $D_{conn} = (graph, boxid, types)$, ahol $graph \in \mathbb{N}$ az algráf azonosító, $boxid \in String$ dobozazonosító, $types \in \langle T_\tau \rangle$ a csatornák típusainak véges sorozata, $types = \langle t_0, t_1, \ldots, t_{n-1} \rangle$.

2.16. definíció (Output csatorna leírók). A B. függelék szerinti EBNF leírás alapján definiáljuk az output csatorna leírókat:

- $D_{null} = (null)$,
- $D_{fix} = (type, id)$, ahol $type \in T_\tau$ a csatorna típusa, $id \in \mathbb{N}_+$ a csatorna azonosítója,

- $D_{connBox} = (thr, boxid, start, types)$, ahol $thr \in \mathbb{H}$ szálazonosító, $boxid \in String$ dobozazonosító, $start \in \mathbb{N}$ csatlakozás kezdőpontját leíró érték, $types \in \langle T_\tau \rangle$ a csatornák típusai, $types = \langle t_0, t_1, \ldots t_{n-1} \rangle$.
- $D_{startGraph} = (sid, count, boxid, types)$, ahol $sid \in \mathbb{N}$ az indítandó algráf azonosítója, $count \in \mathbb{N}_+$ az indítási darabszámot leíró kifejezés, $boxid \in String$ dobozazonosító, $types \in \langle T_\tau \rangle$ a csatornák típusainak véges sorozata, $types = = \langle t_0, t_1, \ldots, t_{n-1} \rangle$.

2.10. jelölés (Típus ellenőrzés). Értelmezzük valamely d input vagy output csatorna leíró és D csatorna leíró típus esetén a $d \simeq D$ műveletet, amely megadja, hogy a d struktúra a D csatorna leíró típusába esik-e.

2.16. megjegyzés. Pl. a $d \simeq D_{fix}$ vizsgálat megállapítja, hogy a d leíró D_{fix} típusú-e.

2.17. megjegyzés. A D_{null} és D_{fix} leírók már az input leírók között is értelmezésre kerültek, de ezek a leírók output esetben is használhatóak. Ennek megfelelően a fenti felsorolásba bekerültek, alakjuk egyezik az input esetben megadottakkal.

2.18. megjegyzés. A $D_{startGraph}$ esetén a *count* elem egy Clean nyelvi kifejezés, mely futás közben kiértékelhető módon állítja elő az adott pozitív egész értéket. A statikus vizsgálat szempontjából ezen értéket 1-nek tekintjük, de figyelembe vesszük, hogy az értéke lehet ettől eltérő is.

2.11. jelölés (D-Box sorozat). Jelöljük $\mathbb{D}$-vel $\langle D_{box} \rangle$ sorozatokat, vagyis a D-Box nyelvi definíciók tetszőleges, nem üres, véges sorozatait.

2.17. definíció (D-Box projekt). Valamely $D_p \in \mathbb{D}$ konkrét D-Box definíciósorozatot *D-Box projektnek* nevezzük.

2.19. megjegyzés. A D-Box projektet D-Box definíciók alkotják. Valójában a fordítóprogram működéséhez, a kód generálásához a D-Box definíciókon kívül szükség lehet a külső típusdeklarációkra (rekord típus) is. Amennyiben egy D-Box programban ilyen típusdeklarációk is szerepelnek, úgy ezek a típusok szerepelnek a projekthez tartozó τ' kezelhető típusok között.

2.5. Csatornahelyesség vizsgálata

Egy adott D-Box projekt esetén vizsgálnunk kell, hogy a projektben szereplő csatornaleírók, és a választott protokollok kapcsolata megfelelő-e. Például egy *memory* protokoll esetén nem lehet egyetlen csatornánk sem, a *joink* protokoll csak egyforma típusú csatornákkal képes működni, stb.

A vizsgálatokhoz először is definiálunk csatornaleíró rekordokat, melyek a csatornák azon jellemzőit tartalmazzák, amelyek a jelen statikus szemantikai vizsgálatokhoz szükségesek. Ezek után a dobozok input és output csatornaleíróit megvizsgálva előállítjuk az adott doboz csatornaleíró rekordjait. Definiálunk olyan ellenőrző függvényeket, mely adott protokoll, és adott leíró rekordok alapján eldönti, hogy a protokollt értelmezhetjük-e ezeken a csatornákon.

2.5.1. A csatornadefiniáló kifejezés felépítése

2.18. definíció (CSATORNALEÍRÓ REKORD). Az $R(t, b)$ formális kettest *csatornaleíró rekordnak* hívjuk, ahol $t \in T_\tau$ a csatorna típusa, $b \in \mathbb{B}$ flag-tipusú bejegyzés, megmutatja hogy statikus vagy dinamikus csatornát jellemzünk-e.

2.19. definíció (CSATORNADEFINIÁLÓ KIFEJEZÉS). Az $f\colon D \to \langle R(t, b)\rangle$ alakú függvényeket *csatornadefiniáló kifejezéseknek* nevezzük, ahol

$$D = \{D_{null}, D_{fix}, D_{auto}, D_{conn}, D_{connBox}, D_{startGraph}\},$$

a függvény argumentuma valamely konkrét csatornadefiniáló kifejezés.

2.20. definíció (NULL CSATORNA). Az $f_{null}\colon D_{null} \to \langle R(t, b)\rangle$, $f_{null}(d) = r$ függvényt *null csatornadefiniáló* kifejezésnek nevezzük, ahol $r \coloneqq \emptyset$.

2.20. megjegyzés. Az f_{null} kifejezés nulla darab csatornaleíró rekordot generál.

2.21. definíció (FIX CSATORNA). Az $f_{fix}\colon D_{fix} \to \langle R(t, b)\rangle$, $f_{fix}(d) = r$ függvényt *fixChannel csatornadefiniáló kifejezésnek* nevezzük, ahol $r \coloneqq \langle r_0\rangle$ csatornaleírók egyelemű sorozata, $r_0 \coloneqq (d.type, false)$.

2.21. megjegyzés. A f_{fix} paramétere egy $d = (type, id)$ leíró. Egyetlen csatornaleírót generál, melynek típusa adott, és statikus jellegű.

2.22. definíció (AUTO CSATORNA). Az $f_{auto}\colon D_{auto} \to \langle R(t, p)\rangle$, $f_{auto}(d) = r$ függvényt *autoChannel csatornadefiniáló kifejezésnek* nevezzük, ahol $r \coloneqq \langle r_0\rangle$ egyelemű sorozat, $r_0 \coloneqq (d.type, false)$.

2.22. megjegyzés. Az f_{auto} paramétere egy $d = (type, auto)$ leíró. Egyetlen csatornaleírót generál, melynek típusa a leíróban szereplővel egyező, és statikus jellegű.

2.23. definíció (AUTOCONNBOX CSATORNA). Legyen $f_{conn}\colon D_{conn} \to \langle R(t, p)\rangle$, $f_{conn}(d) = r$. Ezt a függvényt *autoConnBox csatornadefiniáló kifejezésnek* nevezzük, ahol ha $d.typeDefList = \langle t_0, t_1, \ldots, t_{n-1}\rangle$, akkor $r \coloneqq \langle r_0, r_1, \ldots, r_{n-1}\rangle$, és $\forall i \in [0, n-1]$ esetén $r_i \coloneqq (t_i, true)$.

2.23. megjegyzés. Az *autoConnBox* leíró dinamikus csatornákat jelez, hiszen ezek tényleges darabszáma majd csak futás közben derül ki. A leíró rekordok szempontjából a darabszámot 1-nek tekintjük, vagyis pontosan annyi leíró rekordot állítunk elő, ahány típus szerepel a típuslistán. A leíró rekordok átveszik rendre a típuslistában szereplő típusokat, és a $b = true$ módon jelzik, hogy ezek dinamikus csatornákat jellemeznek.

2.5.2. Input protokoll kifejezések

Vizsgálni kell, hogy az adott doboz input csatornáinak száma és típusa esetén az adott protokoll értelmezve van-e. Például a *joink* csak egyforma típusú csatornák esetén van értelmezve, míg a *join1* eltérő típusú csatornákon is. A protokollok esetén az is fontos, hogy adott számú és típusú csatorna esetén a konkrét protokoll milyen számú és típusú eredményt állít elő. Ezt az eredményt, a protokoll függvény visszatérési értékét össze kell tudni hangolni a számítási kifejezés paraméterezésével. Ezt a feladatot az input wrapper függvény végzi, amely protokoll függvény értékeit a helyreállítható értékek sorozatával összefésülve képezi a kifejezés paramétereit.

2.24. definíció (PROTOKOLL KIFEJEZÉS). Az $f\colon \langle R(t,p)\rangle \to \langle T_{\tau'}\rangle$ alakú függvényeket *protokoll kifejezéseknek* nevezzük.

2.25. definíció (MEMORY KIFEJEZÉS). Az $f_{memory}\colon \langle R(t,p)\rangle \to \langle T_{\tau'}\rangle$, $f_{memory}(R) = = T$ függvényt *memory protokoll kifejezésnek* nevezünk, ahol ha $R = \emptyset$, akkor $T \coloneqq \emptyset$.

2.24. megjegyzés. A *memory* input protokoll esetén a csatornaleíró rekordok listája üres kell legyen, és az eredmény is egy üres lista lesz.

2.26. definíció (JOIN1 KIFEJEZÉS). Az $f_{join1}\colon \langle R(t,p)\rangle \to \langle T_{\tau'}\rangle$, $f_{join1}(R) = T$ függvényt *join1 protokoll kifejezésnek* nevezzük. Ha $R = \langle R_0, R_1, \ldots, R_{n-1}\rangle$, $sizeof(R) > 0$, és $\forall i \in [0, n-1]$-re $R_i.b = false$, akkor $T \coloneqq \langle T_0, T_1, \ldots, T_{n-1}\rangle$, $\forall i \in [0, n-1]$ esetén $T_i \coloneqq R_i.t$.

2.25. megjegyzés. A *join1* input protokoll kifejezés esetén a csatornaleíró rekordok listája statikus csatornákat kell hogy tartalmazzon ($b = false$). A kifejezés eredményében szereplő típusok listája számosságban egyezik a csatornák számával, a típusok is rendre egyeznek. Ennek megfelelően a *join1* protokoll kifejezés nem alkalmazható autoConnBox csatornadefiniáló kifejezéssel együtt.

2.27. definíció (JOINK KIFEJEZÉS). Az $f_{joink}\colon \langle R(t,p)\rangle \to \langle T_{\tau'}\rangle$, $f_{joink}(R) = T$ függvényt *joink input protokoll kifejezésnek* nevezünk, ahol ha $R = \langle R_0, R_1, \ldots, R_{n-1}\rangle$, $sizeof(R) > 0$, és $\forall i, j \in [0, n-1]$ esetén $R_i.t = R_j.t$, akkor $T \coloneqq \langle L(R_0.t)\rangle$ egyelemű sorozat, (L a lista-konstruktor).

2.26. megjegyzés. A *joink* input protokoll esetén a csatornaleíró rekordok listája tartalmazhat dinamikus csatornát is (megengedett a $b = true$ is), de minden csatorna típusának azonosnak kell lennie. Az eredmény egy elemű lista, típusa a csatornák közös típusából képzett lista.

2.5.3. Az input protokoll leírása

2.28. definíció (InpProt felépítés). Egy $D \in D_{box}$ D-Box definíció esetén az input protokoll $D.InpProt = (IChannels, IProtocoll)$ formális kettes, ahol

- $IChannels = \langle (f_0, d_0), (f_1, d_1), \ldots, (f_{n-1}, d_{n-1}) \rangle$ az input csatornadefiniáló kifejezések véges sorozata, $\forall i \in [0, n-1]$ esetén $f_i \in \{f_{null}, f_{fix}, f_{auto}, f_{conn}\}$, $d_i \in$ $\in \{D_{null}, D_{fix}, D_{auto}, D_{conn}\}$, d_i a vele párban álló f_i függvény által megkövetelt típusú argumentum,
- $IProtocoll \in \{f_{memory}, f_{join1}, f_{joink1}\}$ input protokoll kifejezések valamelyike.

2.27. megjegyzés. Az $IChannels$ lista csak megfelelő függvény $\leftrightarrow$ argumentum párokból állhat. A D-Box nyelvi leírás az EBNF szerint (értelemszerűen) csak a csatornaleírókat tartalmazza, ám egy adott csatornadefiniáló kifejezéshez egyértelműen hozzárendhető egy csatornadefiniáló függvény.

2.29. definíció (input sorozat). Egy $D \in D_{box}$ esetén $D.InpProt.IChannels =$ $= \langle (f_0, d_0), (f_1, d_1), \ldots (f_{n-1}, d_{n-1}) \rangle$ input csatornákhoz tartozó $RI(D) = f_0(d_0) \oplus$ $\oplus f_1(d_1) \oplus \cdots \oplus f_{n-1}(d_{n-1})$ sorozatot *input csatornaleíró rekordok sorozatának* nevezzük.

2.28. megjegyzés. A sorozat úgy képződik, hogy vesszük a D-Box definícióban szereplő konkrét input csatornadefiniáló kifejezéseket, alkalmazzuk a megfelelő argumentumokat, és a kapott sorozatokat összefűzzük. Így kapjuk meg, hogy hány darab és milyen típusú (és jellegű) input csatorna szükséges a doboz számára.

2.5.4. Az input helyességének ellenőrzése

Az input csatornák és az input protokoll alkalmazhatóságát vizsgáljuk. A helyességéhez szükséges, hogy amennyiben a kifejezés argumentumainak típuslistájából elhagyjuk a helyreállítható típusokat, akkor pontosan a protokoll függvény által előállított értékek típusait kapjuk meg. Ez azt jelenti, hogy a wrapper függvény a D-Box szempontok szerint képes előállítani a helyes paraméterezést, mivel csak a helyreállítható értékek hiányoznak a paraméterlistáról.

2.12. jelölés (doboz input protokollja). Valamely $D \in D_{box}$ D-Box definíció esetén jelöljük $f_{iprot} \in \{f_{memory}, f_{join1}, f_{joink}\}$ módon az aktuális input protokollt ($f_{iprot} = D.InpProt.IProtocoll$).

2.29. megjegyzés. Az f_{iprot} azt a konkrét input protokoll kifejezést jelöli, amely a konkrét D_{box} definícióban szerepel. Ez csak egy rövidebb leírása a dobozdefinícióban szereplő $D.InpProt.IProtocoll$ leírásnak.

2.30. definíció (INPUT HELYES). Egy $D \in D_{box}$ D-Box definíció *az input típusa szerint statikus szemantikailag helyes*, ha $RI(D)$ a doboz által generált input csatornaleíró rekordok sorozata esetén $f_{iprot}(RI(D)) = f_{styp}(D.ExpressionDef.inpTypes)$.

2.30. megjegyzés. Generáljuk a csatornákat a dobozban lévő csatornadefiniáló kifejezések segítségével, alkalmazzuk rá az input protokollt, így megkapjuk azt a típussorozatot, amely a csatornák és a protokoll alapján előáll. Ha ezen típussorozat megegyezik a kifejezés argumentumainak típussorozatával (számban, sorrendben is) – kivéve az esetleges T_δ helyreállítható típusú paramétereket –, akkor a doboz input típusa szerint helyes.

2.31. megjegyzés. Vegyük észre, hogy az input típusa szerinti helyességhez nem ellenőrizzük le, hogy a $D.ExpressionDef.inpTypes$ típusfelsorolás egyezik-e a doboz $D.ExpressionDef.expression$ részében megadott (Clean nyelvi) kifejezés tényleges argumentum típuslistájával. Ezt a D-Box fordító „elhiszi”, nem áll módjában ugyanis a kettő közötti kapcsolatot ellenőrizni. A típushelyesség eldöntéséhez a D-Box definícióban szereplő információkra kell hagyatkoznia.

Ha a D-Box fordító képes lenne a fenti ellenőrzést végrehajtani, akkor a D-Box definícióban szükségtelen lenne a kifejezés paramétereinek típusát felsorolni. Valójában ennek hiányában a D-Box nyelvi szint szintaktikai helyessége nem biztosítja a generált kód Clean nyelvi szintű szintaktikai helyességét.

A D-Box nyelven elvárás a programozótól, hogy a kifejezés paraméterezésének típusát helyesen adja meg. Valójában a D-Box nyelvi definíciókat leggyakrabban a D-Clean fordító generálja, vagy grafikus IDE felületen szerkesztik meg. Az előző esetben a D-Clean fordító működése közben a *DistrStart* kifejezésben szereplő, a felhasználó által definiált függvények és kifejezések típusát levezeti, így képes generálni a D-Box definíciókban a már levezetett eredmények alapján ezt az információt. A D-Box fordító alacsonyabb absztrakciós szinten már nem áll ilyen szoros kapcsolatban a Clean nyelvvel. A D-Box nyelv ily módon kiterjeszthető lehet más *host* programozási nyelv esetére is. Mint látni fogjuk, a kódgenerálás során is külső sablon fájlokat használunk fel elsősorban, így a kiterjeszthetőség a kódgenerelási módszerben is benne rejtőzik.

Ezen absztrakciós szinten központi fogalommá a *csatorna*, a *doboz*, és a *protokoll* vált, és szükségszerűen a *számítási kifejezés* szerepköre és ellenőrzése a felsőbb szintre tolódott át. Amennyiben a kifejezés tényleges típusa eltér a D-Box definícióban feltüntetettől, úgy a generált kód szintaktikai hibás lesz. Ezt a generált kódra alkalmazott konkrét nyelvi fordító fogja a bináris kód előállításának fázisában észlelni. A generált

kód szintaktikai helyessége valójában függ még a kódgenerálás során alkalmazott sablon fájlokban szereplő kódoktól is.

2.5.5. Konkrét output csatornadefiniáló kifejezések

Hasonlóan az input csatornák esetéhez, az output csatornák vizsgálata is elsősorban az alkalmazott protokollal való összehasonlítás miatt szükséges. Az output csatornadefiniáló kifejezések mindegyike egy vagy több csatornát ír le, melyet a doboz futás közben kezelni fog. A *split1* protokoll esetén ezen csatornák típusai különbözőek is lehetnek, míg a *splitk*, *splitf* esetén csak azonos típusú csatornák lehetnek. Ezért először tárgyaljuk, hogy az egyes csatornadefiniáló kifejezések hány darab és milyen típusú csatornákat írnak le, majd adott doboz esetén egyetlen sorozatba fűzzük az általa leírt csatornákat. Ezt a sorozatot kell összehasonlítani majd a konkrétan megadott protokollal.

Az input csatornák esetén megadott 2.20. és a 2.21. definíciókban szereplő *null*, fix csatornadefiniáló kifejezések változatlan formában értelmezve vannak output csatornák esetén is.

2.31. definíció (connBox csatorna). Legyen az $f_{connBox} \colon D_{connBox} \to \langle R(t,b) \rangle$, $f_{connBox}(d) = r$. Ezt a függvényt *connBox csatornadefiniáló kifejezésnek* nevezzük, ahol ha $d.types = \langle t_0, t_1, \ldots, t_{n-1} \rangle$ a csatorna típusai, akkor $r \coloneqq \langle r_0, r_1, \ldots, r_{n-1} \rangle$, és $\forall i \in \in [0, n-1]$ esetén $r_i \coloneqq (t_i, false)$.

2.32. megjegyzés. A *connBox* csatornák futás közben kapcsolódnak fel egy másik doboz adott csatornájára. Típusuk adott, statikus jellemzőjű.

2.32. definíció (startGraph csatorna). Az $f_{startGraph} \colon D_{startGraph} \to \langle R(t,b) \rangle$, $f_{startGraph}(d) = r$ függvényt *startGraph csatornadefiniáló kifejezésnek* nevezzük, ahol ha $d.types = \langle t_0, t_1, \ldots, t_{n-1} \rangle$, akkor $r \coloneqq \langle r_0, r_1, \ldots, r_{n-1} \rangle$ csatornaleíró rekordok sorozata, és $\forall i \in [0, n-1]$ esetén $r_i \coloneqq (t_i, true)$.

2.33. megjegyzés. A *startGraph* csatornák tényleges száma csak futás közben derül majd ki, a *d.count* kifejezés kiértékelése során, ezért a csatornák dinamikus jellemzőkkel bírnak. Típusaik a csatornaleíróban adottak, dinamikus jellemzőjűek.

2.5.6. Az output protokoll leírása

Az alábbiakban tárgyaljuk, melyik konkrét output protokoll milyen csatorna sorozat esetén van értelmezve. Amennyiben a dobozhoz tartozó output csatornák és a protokoll nem összeegyeztethető, úgy az statikus szemantikai hibát jelent. Az alábbiakban megadott protokoll kifejezések által előállított típuslista jelen esetben nem a csatornáról bejött, a protokoll által produkált típusú értékeket jelöli, hanem épp ellenkezőleg:

a protokollt milyen típusú értékekkel kell felparaméterezni, hogy az értékeket a hozzá rendelt csatornák felé képes legyen továbbítani.

2.34. megjegyzés. A 2.25. definícióban megadott f_{memory} protokoll mint output protokoll is szerepelhet. A memory otput protokoll esetén a csatorna-leíró rekordok listája üres kell legyen, és az eredmény is egy üres lista lesz.

2.33. definíció (SPLIT1 KIFEJEZÉS). Az $f_{split1}\colon \langle R(t,p)\rangle \rightarrow \langle T_{\tau'}\rangle$, $f_{split1}(R) = T$ függvényt *split1 output protokoll kifejezésnek* nevezzük, ahol ha $R = \langle R_0, R_1, \ldots, R_{n-1}\rangle$, és teljesül, hogy $n > 0, \forall i \in [0, n-1]$ esetén $R_i.b = false$, akkor $T \coloneqq \langle T_0, T_1, \ldots, T_{n-1}\rangle$, és $\forall j \in [0, n-1]$ esetén $T_j \coloneqq R_j.t$.

2.35. megjegyzés. A split1 output protokoll esetén a csatornaleíró rekordok listája csupa statikus csatornát kell tartalmazzon ($b = false$), az eredmény típusok listája számosságban egyezik a csatornák számával, típusuk is rendre megegyezik.

2.34. definíció (SPLITK KIFEJEZÉS). az $f_{splitk}\colon \langle R(t,p)\rangle \rightarrow \langle T_{\tau'}\rangle$, $f_{splitk}(R) = T$ függvényt *splitk output protokoll kifejezésnek* nevezzük, ahol ha $R = \langle R_0, R_1, \ldots, R_{n-1}\rangle$, és teljesül, hogy $n > 0$, $\forall i, j \in [0, n-1]$ esetén $R_i.t = R_j.t$, akkor $T \coloneqq \langle L(R_0.t)\rangle$ egyelemű lista (L a lista-konstruktor).

2.36. megjegyzés. A splitk output protokoll esetén a csatornaleíró rekordok listája tartalmazhat dinamikus csatornát is ($b = true$), de minden csatorna típusának azonosnak kell lennie. Az eredmény típusok listája egy elemű, típusa az adott (egyező) típusból képzett lista.

2.35. definíció (SPLITF KIFEJEZÉS). Az $f_{splitf}\colon \langle R(t,p)\rangle \rightarrow \langle T_{\tau'}\rangle$, $f_{splitf}(R) = T$ függvényt *splitf output protokoll kifejezésnek* nevezzük, ahol ha $R = \langle R_0, R_1, \ldots R_{n-1}\rangle$, és teljesül, hogy $n > 0$, $\forall i, j \in [0, n-1]$ esetén $R_i.b = false$, $R_i.t = R_j.t$, akkor $T \coloneqq \langle R_0.t\rangle$.

2.37. megjegyzés. A *splitf* output protokoll esetén a csatornaleíró rekordok listája csak statikus csatornát tartalmazhat, ($b = false$), a csatornák típusának megegyezőnek kell lenniük. A splitf a szemantikájából fakadóan nem ad a végeredmény típushoz egy dimenzió mélységet, ellentétben a splitk-val.

2.36. definíció (OUTPROT FELÉPÍTÉSE). Egy $D \in D_{box}$ D-Box definíció esetén az output protokoll $D.OutProt = (OChannels, OProtocoll)$ formális kettes, ahol

- $OChannels = \langle (f_0, d_0), (f_1, d_1), \ldots, (f_{n-1}, d_{n-1})\rangle$ az output csatornadefiniáló kifejezések véges sorozata, $\forall i \in [0, n-1]$, esetén $f_i \in \{f_{null}, f_{fix}, f_{connBox}, f_{startGraph}\}$, $d_i \in \{D_{null}, D_{fix}, D_{connBox}, D_{startGraph}\}$, d_i a vele párban álló f_i függvény által megkövetelt típusú paraméter,

- $OProtocoll \in \{f_{memory}, f_{split1}, f_{splitk}, f_{splitf}\}$ output protokoll kifejezés.

2.38. megjegyzés. Az $OChannels$ lista csak megfelelő függvény $\leftrightarrow$ argumentum párokból állhat. A függvényeket a rögzített sorrendjükben alkalmazva megkaphatjuk a csatornaleíró rekordok sorozatát.

2.37. definíció (OUTPUT SOROZAT). A $D \in D_{box}$ esetén $D.OutProt.OChannels = \langle (f_0, d_0), (f_1, d_1), \ldots, (f_{n-1}, d_{n-1}) \rangle$ output csatornákhoz tartozó $RO(D) = f_0(d_0) \oplus \oplus f_1(d_1) \oplus \cdots \oplus f_{n-1}(d_{n-1})$ sorozatot *output csatornaleíró rekordok sorozatának* nevezzük.

2.39. megjegyzés. A csatornadefiníciók sorozata úgy képződik, hogy a D-Box definícióban szereplő output csatornadefiniáló felsorolás által képzett csatornaleíró rekordokból sorozatot képzünk, így megkapjuk, hány darab és milyen típusú (és jellegű) output csatorna szükséges a doboz számára.

2.5.7. Az output helyességének ellenőrzése

A statikus helyességhez szükséges, hogy a protokoll és a csatornák összeegyeztethetők legyenek. Ugyanakkor a protokollt is össze kell vetni a kifejezés által előállított értékek típusaival. Ezen típuslistából ki kell szűrni a helyreállítható típusokat, mivel azokat nem lehet a csatornák felé továbbítani. Amennyiben a típuslistáról a helyreállítható típusokat eltávolítva pontosan megkapjuk az adott dobozban definiált protokoll függvény elvárt paramétereinek típusait, úgy a doboz képes az értékek továbbítására a csatornákon.

2.13. jelölés (DOBOZ OUTPUT PROTOKOLLJA). Valamely $D \in D_{box}$ D-Box definíció esetén jelöljük $f_{oprot} \in \{f_{memory}, f_{split1}, f_{splitk}, f_{splitf}\}$ módon az aktuális output protokollt ($f_{oprot} = D.OutProt.OProtocoll$).

2.38. definíció (OUTPUT HELYES). Egy $D \in D_{box}$ D-Box definíció *az output típusa szerint statikus szemantikailag helyes*, ha

$$f_{oprot}(RO(D)) = f_{styp}(ExpressionDef.outTypes).$$

2.40. megjegyzés. Generáljuk a csatornákat a felsorolt output csatornadefiniáló kifejezések segítségével, alkalmazzuk rá az output protokoll kifejezést, így megkapjuk azt a típus sorozatot, amely jelöli ezen konkrét esetben az output protokoll paraméterezését. Ha ezen típus sorozat megegyezik a kifejezés output argumentumainak típus sorozatával (számban, sorrendben is, kivéve az esetleges δ helyreállítható típusú paramétereket), akkor a doboz output része helyes.

2.41. megjegyzés. Hasonlóan az input rész helyességének ellenőrzésekor, most sem ellenőrizzük le, hogy a $D.ExpressionDef.outTypes$ típusfelsorolás valóban egyezik-e a $D.ExpressionDef.expression$ részben megadott (Clean nyelvi) kifejezés eredményének típuslistájával. Ezt a D-Box fordító *elhiszi*, nem áll módjában ugyanis a kettő közötti kapcsolatot ellenőrizni. A típushelyesség ezen szintű eldöntéséhez a D-Box definícióban szereplő információkra kell hagyatkoznia.

2.6. A csatornahivatkozások ellenőrzése

Az előzőekben megfogalmaztuk, hogy a csatornákra pontosan két ponton szabad egy projekten belül hivatkozni. Az egyik hivatkozás szerepköre szerint input csatorna, a másik pedig output csatorna kell legyen. Amennyiben a csatornahivatkozásokat e szempont szerint kívánjuk ellenőrízni, nem szükséges a protokollok (join1, splitf, stb.) figyelembe vétele, mivel azok csak a már meglévő csatornák és a dobozban lévő kifejezés közötti kapcsolat szempontjából érdekesek.

A csatornák használatának keresztellenőrzésére passzív és aktív rekordokat definiálunk. Ezeket a csatornadefiniáló kifejezések példányosítják. Először generáljuk az input, és az output csatornákra is ezeket az ellenőrző rekordokat, majd a kapott sorozatokat összehasonlítjuk.

Az *aktív* rekordok csatlakozáskérést írnak le, más csatornákat *keresnek*. A *passzív* leíró rekordok olyan csatornákat írnak le, amelyek megtalálásra várnak.

Az aktív rekordok információt hordoznak: mely csatornához kívánnak csatlakozni. A cél csatornát vagy a rögzített (fix) csatorna azonosítója alapján hivatkozzák meg, vagy megadják pontosan melyik doboz sorrendben hányadik csatornájára kívánnak csatlakozni. Ezt meg tudják adni input esetben a *autoConnBox* és a *fix*, output esetben a *fix*, *connBox* és *startGraph* csatornadefiniáló kifejezések.

A passzív rekordok információt hordoznak arról, hogy mely doboz sorrendben hanyadik csatornáját képviselik, és amennyiben van rögzített csatorna azonosítójuk, az milyen értékű. Például az *auto* csatornadefiniáló kifejezés képes ilyen passzív rekordot leírni input esetben.

Vegyük észre, hogy az aktív esetben felsorolt csatornadefiniáló kifejezések passzív állapotot is le tudnak írni, hiszen bármely csatornadefiniáló kifejezés esetén passzív rekordok is generálhatóak (ezek a csatornák is tartoznak valamely dobozhoz, a sorrendbe beilleszthetőek). Az *auto* kifejezések azonban semmilyen formában nem lehetnek aktívak, mivel nem tudják leírni azt, hogy melyik dobozra kívánnak csatlakozni.

2.39. definíció (PASSZÍV REKORD). Az $R_p = (boxid, order, type, id)$ formális négyest *passzív ellenőrző rekordnak* nevezzük, ahol

- $boxid \in String$ valamely D-Box definíció BoxID-je,

- $order \in \mathbb{N}$ a dobozon belüli csatornasorrend ($order >= 0$),
- $type \in T_\tau$ a csatorna típusa,
- $id \in \mathbb{N} \cup \{-1, -2, -3\}$ a csatorna feltüntetett azonosítója, amely vagy egy konkrét egész szám ($id > 0$), vagy nem definiált, *auto* jellegű ($id = -1$), vagy tiltott kapcsolódású ($id = -2$), vagy *autoConnBox* csatorna ($id = -3$).

Egy *autoConnBox* input csatorna tisztán passzív szerepkörű ugyan, de egyetlen típusú output csatornából szabad csak rá hivatkozni: a *connBox ancestorThread* típusúból.

A passzív rekordok generálásakor a dobozból csak az *InpProt.IChannels* iletve az *OutProt.OChannels* részt kell használni. A doboz belsejében szereplő kifejezés (*ExpressionDef*) részei, és a konkrét protokoll (*InpProt.IProtocoll*, *OutProt.OProtocoll*) jelen esetben érdektelenek. Ez utóbbiak csak az input/output statikus szemantikai helyesség szempontjából voltak fontosak.

2.40. definíció (AKTÍV REKORD). Az $R_a = (boxid, order, type, id)$ formális négyest valamely csatorna *aktív ellenőrző rekordjának* nevezzük, ahol

- $boxid \in String$ valamely D-Box definíció BoxID-je,
- $order \in \mathbb{N}$ a dobozon belüli csatorna-sorrend ($order >= 0$),
- $type \in T_\tau$ a csatorna típusa,
- $id \in \mathbb{N} \cup \{-1\}$ a csatorna feltüntetett azonosítója, amely vagy egy konkrét egész szám ($id >= 0$), vagy automatikus kiosztású azonosító ($id = -1$).

2.42. megjegyzés. Az aktív rekord nem azt írja le, hogy ezen csatorna melyik dobozhoz tartozik, hanem hogy melyik doboz melyik csatornájára kíván rácsatlakozni. Ezt vagy a dobozon belüli sorszámmal (*order*) adjuk meg, vagy a keresett csatorna egyedi azonosítójával (*id*). Utóbbi esetben a doboz azonosító és sorrend értéke nem fontos.

2.43. megjegyzés. A csatornahivatkozások ellenőrzéséhez négy ellenőrző függvényt definiálunk:

- egy input passzívhoz ellenőrizni kell, hogy pontosan egy output aktív tartozik,
- egy input aktívhoz ellenőrizni kell, hogy pontosan egy output passzív tartozik,
- egy output passzívhoz ellenőrizni kell, hogy pontosan egy input aktív tartozik,
- egy output aktívhoz ellenőrizni kell, hogy pontosan egy input passzív tartozik.

2.44. megjegyzés. Az alábbi összekapcsolódások képzelhetők el:

input		*output*
fix	↔	fix
auto	↔	startGraph
auto	↔	connBox thisThread
autoConnBox	↔	connBox ancestorThread

D-Box definíció

```
1  BOX BoxID_1001
2  {   1, { (([Int],1001)), join1},
3        { (...) },
4        { (...)  }
5  }
6  BOX BoxID_1002  // egyik ráhivatkozó doboz
7  {   1,  { (...},
8        { (...},
9        { ([Int],1001), join-1}
10 }
11 BOX BoxID_1003  // másik ráhivatkozó doboz
12 {   1,  { ( ... },
13       { ( ... },
14       { ([Int],1001), join-1}
15 }
```

2.1. példa. Hibás (többszörös) input csatorna felhasználás ID alapján

Az ellenőrzés során az alábbi hibaforrásokat kell kiszűrni:

- Egy fix azonosítójú input csatornára több doboz output részéről is hivatkozhatunk az azonosítója segítségével direktben (2.1. példa). Ez nem csak input, de output csatorna esetén is elképzelhető.
- Valamely fix vagy auto csatornára dobozon belüli sorrendje alapján több csatorna is hivatkozhat (2.2. példa).

A 2.2. példában szereplő *BoxID_ 1001* doboz mindkét input csatornájára többszörös hivatkozás látható. A *startGraph* kifejezés ezen algráfot indítja el (2-es algráf), és indítás után az algráf ezen dobozaira kíván felcsatlakozni. A *connBox* kifejezésben is direktben ezen doboz van megemlítve, ennek is az első csatornájától kezdve két egymást követő csatornára csatlakozás van leírva. Ezen *connBox* kifejezés egyébként szintaktikai hibás amiatt is, hogy az őt tartalmazó *BoxID_ 1002* doboz, illetve a csatlakozó doboz nem ugyanazon algráfba esik, de ezen ellenőrzésről majd csak a 2.7. fejezetben lesz szó.

2.6.1. Passzív rekordok generálása input csatornákhoz

Minden egyes (*null*, *fix*, *auto*, *conn*) input csatorna leíró alapján megadjuk, hogyan kell képezni a hozzá tartozó passzív rekordokat. Adott doboz esetén a leírók alapján képzett

D-Box definíció

```
1  BOX BoxID_1001
2  {   2, { (([Int],1001),([Int],auto)), join1},
3        { (...) },
4        { (...)  }
5  }
6  BOX BoxID_1002  // egyik ráhivatkozó doboz
7  {   1,  { (...},
8        { (...},
9        { (connBox 1 "BoxID_1001" ([Int],[Int]), join1}
10 }
11 BOX BoxID_1004  // másik ráhivatkozó doboz
12 {   1,  { (...},
13       { (...},
14       { (startGraph 1 2 "BoxID_1001" ([Int],[Int]), join1}
15 }
```

2.2. példa. Hibás (többszörös) input csatorna felhasználás sorrend alapján

passzív rekordokat sorozatba kell fűzni, ahol már a sorszám (*order*) is meghatározható. A projektet alkotó dobozokhoz tartozó sorozatokat egyetlen nagy sorozattá is összefűzzük, így megkapjuk a projektben definiált csatornák passzív rekordjait. Amennyiben képezzük az output csatornákhoz tartozó projekt szintű aktív rekordok sorozatát, úgy ezen két sorozatot megfelelő módon vizsgálva eldönthető, hogy ezen aspektusból a csatornahivatkozások rendben vannak-e.

2.41. definíció (NULL PASSZÍV). Az $fp_{null}\colon D_{box} \times D_{null} \to \langle R_p \rangle, fp_{null}(b,d) = r$ függvényt *null passzív* rekordgeneráló függvénynek nevezzük, ahol $r \coloneqq \emptyset$.

2.45. megjegyzés. A *null* input csatorna leíró nem generál csatornát, így passzív ellenőrző rekord sem rendelhető hozzá. Az eredmény egy üres sorozat.

2.42. definíció (FIX PASSZÍV). Az $fp_{fix}\colon D_{box} \times D_{fix} \to \langle R_p \rangle, fp_{fix}(b,d) = r$ függvényt *fix passzív* rekordgeneráló függvénynek nevezzük, ahol $r \coloneqq \langle r_0 \rangle$ egyelemű sorozat, $r_0 \coloneqq (b.BoxID, 0, d.type, d.id)$.

2.46. megjegyzés. A *fix* csatorna leíró egyetlen csatornát ír le, így egyetlen passzív rekordot generál. Ezen rekord tartalmazza melyik dobozhoz tartozik (b.BoxID), a csatorna típusát és azonosítóját is. A rekord *order* része egyenlőre nem kitölthető, ezért azt 0-ra állítjuk be.

2.43. definíció (AUTO PASSZÍV). Az $fp_{auto}\colon D_{box} \times D_{auto} \to \langle R_p \rangle, fp_{auto}(b,d) = r$ függvényt *auto passzív* rekordgeneráló függvénynek nevezzük, ahol $r \coloneqq \langle r_0 \rangle$ egyelemű sorozat, $r_0 \coloneqq (b.BoxID, 0, d.type, -1)$.

2.47. megjegyzés. Az *auto* csatorna leíróban a típus ismert, de a csatorna azonosítója csak futás közben fog kiderülni. A passzív rekordba ezért bekerül a doboz azonosítója, és a típus. Az *id* helyére a -1 érték kerül, mely jelzi hogy ez egy dinamikus azonosítóval ellátott csatorna (emiatt nem lehet rá majd csak sorrend alapján hivatkozni).

2.44. definíció (AUTOCONNBOX PASSZÍV). Legyen az $fp_{conn}\colon D_{box} \times D_{conn} \to \langle R_p \rangle$, $fp_{auto}(b,d) = r$. Ezt a függvényt *autoConnBox passzív* rekordgeneráló függvénynek nevezzük, ahol ha $d.types = \langle t_0, t_1, \ldots, t_{n-1} \rangle$, akkor $r \coloneqq \langle r_0, r_1, \ldots, r_{n-1} \rangle$, és $\forall i \in \in [0, n-1]$ esetén $r_i \coloneqq (b.BoxID, 0, t_i, -3)$.

2.48. megjegyzés. Egy *autoConnBox* csatorna leíró több csatornát képes definiálni, ezért a képzett passzív rekord sorozat (általában) nem egy elemű. A képzett sorozat elemszáma megegyezik a *types* részben felsorolt típusok számával, mivel ez jelzi, hogy hány csatornára kívánunk rácsatlakozni a célként megjelölt dobozban. A rekordok tartalmazzák a doboz azonosítóját, a típust, és az azonosító helyén a -3 értéket, jelezvén hogy ez egy *autoConnBox* kifejezés által leírt passzív rekord. Erre csak korlátozott módon lehet aktívan hivatkozni.

2.45. definíció (DOBOZ INPUT PASSZÍV). Az $fp_{DI}\colon D_{box} \to \langle R_p \rangle$, $fp_{DI}(b) = r$ függvényt doboz szintű *input passzív* rekordgeneráló függvénynek nevezzük, ahol ha

$$b.InpProt.IChannels = \langle (f_0, d_0), (f_1, d_1), \ldots, (f_{n-1}, d_{n-1}) \rangle$$

csatorna definiáló kifejezések, akkor

$$r \coloneqq fp_0(d_0) \oplus fp_1(d_1) \oplus \cdots \oplus fp_{n-1}(d_{n-1})$$

az input passzív rekordgeneráló függvények által előállított rekordsorozatok füzére, ahol ha $r = \langle r_0, r_1, \ldots, r_{m-1} \rangle$, akkor $\forall i \in [0, m-1]$ esetén $r_i.order \coloneqq i$. (Az r sorozat képzése során $\forall i \in [0, n-1]$ esetén $fp_i \coloneqq fp_{null}$ ha $f_i = f_{null}$, $fp_i \coloneqq fp_{fix}$ ha $f_i = f_{fix}$, $fp_i \coloneqq fp_{auto}$ ha $f_i = f_{auto}$, $fp_i \coloneqq fp_{conn}$ ha $f_i = f_{conn}$.)

2.49. megjegyzés. A fenti doboz szintű passzív rekordgeneráló függvény már egyben látja a dobozban definiált csatornákat, így ki tudja osztani a passzív rekordokban szereplő sorrend (order) értékeket. Mellékesen a passzív rekordgeneráló függvények sorozatait egyetlen sorozattá egyesíti. Az összesítés során a sorrend fontos, mert az order mező kitöltése folyamatos sorszámozással történik (0-tól induló sorszámok).

2.46. definíció (PROJEKT INPUT PASSZÍV). Az $fp_{PI}\colon \mathbb{D} \to \langle R_p \rangle$, $fp_{PI}(D_p) = r$ függvényt *projekt szintű input passzív* rekordgeneráló függvénynek nevezzük, ahol ha

$$D_p = \langle D_0, D_1, \ldots, D_{m-1} \rangle,$$

akkor

$$r \coloneqq fp_{DI}(D_0) \oplus fp_{DI}(D_1) \oplus \cdots \oplus fp_{DI}(D_{m-1}).$$

2.50. megjegyzés. A projekt szintű generáló függvény a doboz szintű passzív rekordgeneráló függvények sorozatait összefűzi, így kapjuk meg a projektben szereplő összes input csatorna passzív rekordjait egyetlen nagy sorozatban. Az összesítés során valójában lényegtelen, hogy milyen sorrendben dolgozzuk fel a doboz definíciókat.

2.6.2. Passzív rekordok generálása output csatornákhoz

Az input esethez hasonlóan az output csatornadefiniáló kifejezésekhez is megadható, milyen passzív leíró rekordot állítanak elő. Ezen passzív leírókat először doboz szinten kell összesíteni, ekkor kitölthetők a sorrendi értékek is, majd projekt szinten is összesíthetőek. A kapott projekt szintű sorozatot a projekt szintű input aktív leírókkal kell majd összevetni, és elemezni a kapcsolatokat.

2.51. megjegyzés. A 2.41. és a 2.42. definíciókban szereplő fp_{null} és fp_{fix} függvények változatlanul használhatóak output csatornák passzív rekordgeneráló függvényeiként is.

2.47. definíció (CONNBOX PASSZÍV). Legyen az $fp_{connBox}\colon D_{box} \times D_{connBox} \to \langle R_p \rangle$, $fp_{connBox}(b, d) = r$. Ezt a függvényt *connBox passzív* rekordgeneráló függvénynek nevezzük, ahol ha $d.types = \langle t_0, t_1, \ldots t_{n-1} \rangle$, akkor $r \coloneqq \langle r_0, r_1, \ldots, r_{n-1} \rangle$, és $\forall i \in [0, n-1]$ esetén $r_i \coloneqq (b.BoxID, 0, t_i, -2)$.

2.52. megjegyzés. A *connBox* esetben annyi passzív rekord keletkezik, ahány elemű a *types* lista volt. A rekordok tartalmazzák a doboz azonosítót, és a csatorna azonosító helyén a -2 értéket. Ez jelzi majd, hogy ezen csatornákra aktív módon tilos hivatkozni.

2.48. definíció (STARTGRAPH PASSZÍV). Az $fp_{startGraph}\colon D_{box} \times D_{startGraph} \to \langle R_p \rangle$, $fp_{startGraph}(b, d) = r$ függvényt *startGraph passzív* rekordgeneráló függvénynek nevezzük, ahol ha $d.types = \langle t_0, t_1, \ldots t_{n-1} \rangle$, akkor $r \coloneqq \langle r_0, r_1, \ldots, r_{n-1} \rangle$, és $\forall i \in [0, n-1]$ esetén $r_i \coloneqq (b.BoxID, 0, t_i, -2)$.

2.53. megjegyzés. Futás közben a csatornákból *d.count* kollekció készül el. A statikus ellenőrzés során feltételezzük, hogy 1 kollekció készül. Ezen feltételezés mellett kell jól működnie az aktív-passzív párosításoknak. Egy későbbi ponton majd ellenőrizni fogjuk, hogy a *startGraph* csatornák az általuk indított algráfok dobozainak *auto* csatornáihoz kapcsolódnak-e – ez biztosítja majd, hogy tetszőleges *d.count* érték mellett is jól működjön a projekt.

2.49. definíció (DOBOZ OUTPUT PASSZÍV). Az $fp_{DO}\colon D_{box} \to \langle R_p\rangle, fp_{DO}(b) = r$ függvényt doboz szintű *output passzív* rekordgeneráló függvénynek nevezzük, ahol ha

$$b.OutProt.OChannels = \langle (f_0, d_0), (f_1, d_1), \ldots, (f_{n-1}, d_{n-1})\rangle$$

csatorna definiáló kifejezések, akkor

$$r \coloneqq fp_0(d_0) \oplus fp_1(d_1) \oplus \cdots \oplus fp_{n-1}(d_{n-1})$$

az output passzív rekordgeneráló függvények által előállított rekordsorozatok füzére, ahol ha $r = \langle r_0, r_1, \ldots, r_{m-1}\rangle$, akkor $\forall i \in [0, m-1]$ esetén $r_i.order \coloneqq i$. ($fp_i \coloneqq \coloneqq fp_{null}$ ha $f_i = f_{null}$, $fp_i \coloneqq fp_{fix}$ ha $f_i = f_{fix}$, $fp_i \coloneqq fp_{connBox}$ ha $f_i = f_{connBox}$, $fp_i \coloneqq fp_{startGraph}$ ha $f_i = f_{startGraph}$).

2.54. megjegyzés. A fenti doboz szintű passzív rekordgeneráló függvény már egyben látja a dobozban definiált csatornákat, így fő feladata a passzív rekordokban szereplő sorrend (order) értékek kiosztása, illetve a megfelelő generált sorozatok összefűzése.

2.50. definíció (PROJEKT OUTPUT PASSZÍV). Az $fp_{PO}\colon \mathbb{D} \to \langle R_p\rangle$, $fp_{PI}(D_p) = r$ függvényt *projekt szintű output passzív* rekordgeneráló függvénynek nevezzük, ahol ha

$$D_p = \langle D_0, D_1, \ldots, D_{m-1}\rangle,$$

akkor

$$r \coloneqq fp_{DO}(D_0) \oplus fp_{DO}(D_1) \oplus \cdots \oplus fp_{DI}(D_{m-1}).$$

2.55. megjegyzés. A projekt szintű generáló függvény a doboz szintű sorozatokat fűzi össze.

2.6.3. Aktív rekordok generálása input csatornákhoz

Az aktív rekordok agresszívebb viselkedésűek, mint a passzív rekordok – azt írják le, kihez kívánnak kapcsolódni. Először megadjuk, melyik input csatorna definiáló kifejezéshez milyen aktív rekord tartozik, majd ezeket doboz szinten sorozattá egyesítjük, végül projekt szinten is összefűzzük. A projekt szintű aktív sorozatbeli elemek a projekt szintű passzív output sorozatbeli elemekkel kerül majd a későbbiekben összepárosításra.

2.51. definíció (NULL AKTÍV). Az $fa_{null}\colon D_{box} \times D_{null} \to \langle R_a\rangle$, $fa_{null}(b, d) = r$ függvényt *null aktív* rekordgeneráló függvénynek nevezzük, ahol $r \coloneqq \emptyset$.

2.56. megjegyzés. A *null* leíró nem definiál aktív rekordot.

2.52. definíció (FIX AKTÍV). Az $fa_{fix}\colon D_{box} \times D_{fix} \to \langle R_a \rangle$, $fa_{fix}(b,d) = r$ függvényt *fix aktív* rekordgeneráló függvénynek nevezzük, ahol $r \coloneqq \langle r_0 \rangle$ egyelemű sorozat, $r_0 \coloneqq (\text{""}, -1, d.type, d.id)$.

2.57. megjegyzés. A *fix* leíró egyetlen aktív rekordot generál. A hivatkozás a csatornára direktben történik az azonosítója alapján, miközben nem ismert, hogy melyik doboz hányadik csatornája ez. Ezért sem a doboz azonosító, sem az *order* nincs kitöltve. Az *order* értéke azért -1, mert a passzív rekordok *order* értéke 0-tól induló sorszámok. Így kizárjuk a párosítás során a véletlen egyezés lehetőségét. A dobozazonosító üres string lesz.

2.53. definíció (AUTO). Az $fa_{auto}\colon D_{box} \times D_{auto} \to \langle R_a \rangle$, $fa_{auto}(b,d) = r$ függvényt *auto aktív* rekordgeneráló függvénynek nevezzük, ahol $r \coloneqq \emptyset$.

2.58. megjegyzés. Az *auto* leíró nem képes aktív rekordot generálni.

2.54. definíció (AUTOCONNBOX AKTÍV). Legyen az $fa_{conn}\colon D_{box} \times D_{conn} \to \langle R_a \rangle$, $fa_{conn}(b,d) = r$. Ezt a függvényt *conn aktív* rekordgeneráló függvénynek nevezzük, ahol ha $d.types = \langle t_0, t_1, \ldots, t_{n-1} \rangle$, akkor $r \coloneqq \langle r_0, r_1, \ldots, r_{n-1} \rangle$, és $\forall i \in [0, n-1]$ esetén $r_i \coloneqq$ $\coloneqq (d.BoxID, i, d.type, -1)$.

2.59. megjegyzés. Az *autoConnBox* esetben aktív rekordok sorozatát lehet generálni, ahol ismert a cél doboz azonosítója. A doboz csatornáira a 0. sorszámtól kezdünk el felcsatlakozni, ezért az *order* kiosztása 0-tól indul. Az $id = -1$, mivel nincs rögzített *id*-je a keresett csatornáknak.

2.55. definíció (DOBOZ INPUT AKTÍV). Az $fa_{DI}\colon D_{box} \to \langle R_a \rangle$, $fa_{DI}(b) = r$ függvényt *doboz szintű input aktív* rekordgeneráló függvénynek nevezzük, ahol ha

$$D.InpProt.IChannels = \langle (f_0, d_0), (f_1, d_1), \ldots, (f_{n-1}, d_{n-1}) \rangle,$$

akkor

$$r \coloneqq fp_0(d_0) \oplus fp_1(d_1) \oplus \cdots \oplus fp_{n-1}(d_{n-1})$$

az input aktív rekordgeneráló függvények által előállított rekordsorozatok füzére. (Az $fa_i \coloneqq fa_{null}$ ha $f_i = f_{null}$, $fa_i \coloneqq fa_{fix}$ ha $f_i = f_{fix}$, $fa_i \coloneqq fa_{auto}$ ha $f_i = f_{auto}$, $fa_i \coloneqq fa_{conn}$ ha $f_i = f_{conn}$.)

2.56. definíció (PROJEKT INPUT AKTÍV). Az $fa_{PI}\colon \mathbb{D} \to \langle R_a \rangle$, $fa_{PI}(D_p) = r$ függvényt *projekt szintű input aktív* rekordgeneráló függvénynek nevezzük, ahol ha $D_p = \langle D_0, D_1, \ldots D_{n-1} \rangle$, akkor $r \coloneqq fa_{DI}(D_0) \oplus fa_{DI}(D_1) \oplus \cdots \oplus fa_{DI}(D_{n-1})$.

2.60. megjegyzés. A doboz szintű input aktív generáló függvény egy konkrét doboz esetén generálja le az összes aktív leíró rekordot. A projekt szintű függvény minden dobozra alkalmazza a doboz szintű függvényt, előállítván a projektben szereplő összes input csatorna aktív leíró rekordjait.

2.6.4. Aktív rekordok generálása output csatornákhoz

Az output csatornák aktív rekordjait is generálni kell a későbbi párosítási ellenőrzések támogatásához. Az aktív rekordok képzését először az adott csatorna definiáló kifejezések esetén vizsgáljuk. Majd módszert adunk, hogyan kell ezeket doboz szinten, majd projekt szinten összesíteni.

2.61. megjegyzés. Az input esetben megismert fa_{null}, fa_{fix} függvények változatlan formában használhatóak output esetekben is.

2.57. definíció (connBox aktív). Legyen az $fa_{connBox}\colon D_{box} \times D_{connBox} \to \langle R_a \rangle$, $fa_{connBox}(b,d) = r$. Ezt a függvényt *connBox aktív* rekordgeneráló függvénynek nevezzük, ahol ha $d.types = \langle t_0, t_1, \ldots, t_{n-1} \rangle$, akkor $r \coloneqq \langle r_0, r_1, \ldots, r_{n-1} \rangle$, ha $d.thr = = thisThread$ akkor $xid \coloneqq -1$, különben $xid \coloneqq -2$, $\forall i \in [0, n-1]$ esetén $r_i \coloneqq \coloneqq (b.BoxID, start + i - 1, t_i, xid)$.

2.62. megjegyzés. A D-Box nyelvi definícióban szereplő *connBox* kifejezésben a *start* értéke a csatornák sorszámát adja meg, de 1-től indul a sorszámozás. Az ellenőrző rekordok generálása során az *order* értéke mindig 0-tól van számozva, ezért kell 1-et levonni (*start+i-1*). Ha a hivatkozott doboz ugyanebben a szálban fut ($thisThread$), akkor -1-es (auto) típusú csatorna lehet csak a hivatkozott. Ha $ancestorThread$, akkor -2-es (autoConnBox) csatorna lehet.

2.58. definíció (startGraph aktív). Az $fa_{startGraph}\colon D_{box} \times D_{startGraph} \to \langle R_a \rangle$, $fa_{connBox}(b,d) = r$ függvényt *startGraph aktív* rekordgeneráló függvénynek nevezzük, ahol ha $d.types = \langle t_0, t_1, \ldots, t_{n-1} \rangle$, akkor $r \coloneqq \langle r_0, r_1, \ldots, r_{n-1} \rangle$, $\forall i \in [0, n-1]$ esetén $r_i \coloneqq (b.BoxID, i, t_i, -1)$.

2.63. megjegyzés. A *startGraph* csak -1-es id-jű (auto) csatornákra képes csatlakozni. A kiválasztott doboz nulladik sorszámú csatornájától indul a felcsatlakozás.

2.59. definíció (doboz output aktív). Az $fa_{DO}\colon D_{box} \to \langle R_a \rangle$, $fa_{DO}(b) = r$ függvényt *doboz szintű output aktív* rekordgeneráló függvénynek nevezzük, ahol ha

$$D.OutProt.OChannels = \langle (f_0, d_0), (f_1, d_1), \ldots, (f_{n-1}, d_{n-1}) \rangle,$$

akkor

$$r \coloneqq fa_0(d_0) \oplus fa_1(d_1) \oplus \cdots \oplus fa_{n-1}(d_{n-1})$$

az output aktív rekordgeneráló függvények által előállított rekord-sorozatok füzére. ($fa_i = fa_{null}$ ha $f_i = f_{null}$, $fa_i = fa_{fix}$ ha $f_i = f_{fix}$, $fa_i = fa_{connBox}$ ha $f_i = f_{connBox}$, $fa_i = fa_{startGraph}$ ha $f_i = f_{startGraph}$.)

2.60. definíció (PROJEKT OUTPUT AKTÍV). Az $fa_{PO}\colon \mathbb{D} \to \langle R_p \rangle$, $fa_{PO}(D_p) = r$ függvényt *projekt szintű output aktív* rekordgeneráló függvénynek nevezzük, ahol $r \coloneqq fa_{DO}(D_0) \oplus fa_{DO}(D_1) \oplus \cdots \oplus fa_{DO}(D_{m-1})$.

2.64. megjegyzés. A doboz szintű output aktív generáló függvény egy konkrét doboz esetén generálja le az összes aktív leíró rekordot. A projekt szintű függvény minden dobozra alkalmazza a doboz szintű függvényt, előállítván a projektben szereplő összes output csatorna aktív leíró rekordját.

2.6.5. Passzív → aktív párkereső függvények

Amennyiben adva van egy projekt szintű passzív leíró rekordok sorozata, és egy aktív leíró rekordsorozat is, ellenőrizni kell a párosíthatóságot. A passzív rekordok aspektusából nézve minden passzív rekordhoz léteznie kell pontosan egy aktív rekordnak, ami rá hivatkozik. A hivatkozás történhet csatorna azonosító, vagy dobozon belüli sorrendi alapon. Az alábbiakban megadott párkereső függvények adott, konkrét passzív rekordhoz kikeresik az aktív párokat.

2.61. definíció (ID SZERINT P→A). Az $fpa_{id}\colon R_p \times \langle R_a \rangle \to \mathcal{P}(R_a)$, $fpa_{id}(P, S) = r$ függvényt *id szerinti passzív-aktív* párkereső függvénynek nevezzük, ahol

- ha $P.id \geqq 0$, akkor $r \coloneqq \{a\colon a \in S, a.id = P.id\}$
- különben $r \coloneqq \emptyset$.

2.65. megjegyzés. Ezen függvény egyszerű: meg kell keresni az összes olyan aktív rekordot, ahol ugyanez az *id* szerepel. Jegyezzük meg, hogy a két oldal típusát nem egyeztetjük, mivel egyenlőre az aktív párok halmazának számossága az érdekes.

2.62. definíció (SORREND SZERINTI P→A). Legyen $fpa_{order}\colon R_p \times \langle R_a \rangle \to \mathcal{P}(R_a)$, $fpa_{order}(P, S) = r$. Ezt a függvényt *sorrend szerinti passzív-aktív* párkereső függvénynek nevezzük, ahol

- ha $P.id \geqq 0$, akkor $r \coloneqq \emptyset$
- különben $r \coloneqq \{a\colon a \in S, a.order = P.order \wedge a.boxid = P.boxid\}$

2.66. megjegyzés. Azon aktív párokat keressük, amelyek ugyanezen doboz (*boxid*) sorrend szerint (*order*) ugyanezen rekordjára hivatkoznak. Szintén nem foglalkozunk azzal, hogy az aktív oldalon ugyanezen típus van-e megadva, hiszen egyenlőre a halmaz számossága lesz az érdekes.

2.6.6. Aktív → passzív párkereső függvények

Hasonlóan a passzív → aktív párkeresés esetén, az aktív rekordok szemszögéből is ki kell keresni az általuk hivatkozott passzív rekordokat. A hivatkozás itt is csatorna azonosító, vagy dobozon belüli sorszám alapján történhet.

2.63. definíció (ID SZERINTI A→P). Az $fap_{id}\colon R_a \times \langle R_p\rangle \to \mathcal{P}(R_p)$, $fap_{id}(A,S) = = r$ függvényt *id szerinti aktív-passzív* párkereső függvénynek nevezzük, ahol

- ha $A.id \geqq 0$, akkor $r \coloneqq \{p\colon p \in S \wedge p.id = A.id\}$
- különben $r \coloneqq \emptyset$.

2.64. definíció (SORREND SZERINTI A→P). Egy $fap_{order}\colon R_a \times \langle R_p\rangle \to \mathcal{P}(R_p)$, $fap_{order}(A,S) = r$ függvényt *sorrend szerinti aktív-passzív* párkereső függvénynek nevezzük, ahol

- ha $A.id \geqq 0$, akkor $r \coloneqq \emptyset$
- különben $r \coloneqq \{p\colon p \in S, p.order = A.order \wedge p.boxid = A.boxid\}$

2.6.7. Passzív → aktív ellenőrző függvények

A passzív→ aktív ellenőrző függvények a párkeresés eredményét értékelik ki. A párkeresés eredményes, ha minden passzívhoz pontosan egy aktív hivatkozás van, és a hivatkozó aktív rekord is ugyanezen csatornatípust feltételez. További ellenőrzések is jelen vannak, adott jellegű passzív rekordra csak adott jellegű aktív rekordok hivatkozhatnak. A részleteket a 2.44. megjegyzés, illetve a 2.67. megjegyzés tartalmazza.

2.65. definíció (P→A ELLENŐRZŐ). Egy $f_{chkPA}\colon R_p \times \langle R_a\rangle \to \mathbb{B}$, $f_{chkPA}(P,S) = = B$ függvényt *passzív-aktív csatornakapcsolat-ellenőrző* függvénynek nevezzük, ahol $T \coloneqq fpa_{id}(P,S) \cup fpa_{order}(P,S)$, $T = \langle a_0, a_1, \ldots, a_{n-1}\rangle$ esetén

- ha $P.id = -3$ akkor ha $sizeof(T) = 1 \wedge a_0.id = -2 \wedge a_0.type = P.type$ akkor $B \coloneqq true$
- ha $P.id = -2$ akkor ha $sizeof(T) = 0$ akkor $B \coloneqq true$
- ha $P.id = -1$ akkor ha $sizeof(T) = 1 \wedge a_0.id < 0 \wedge a_0.id = -1 \wedge a_0.type = P.type$ akkor $B \coloneqq true$
- ha $P.id \geqq 0$ akkor ha $sizeof(T) = 1 \wedge P.id = a_0.id \wedge a_0.type = P.type$ akkor $B \coloneqq true$
- minden más esetben $B \coloneqq false$.

2.67. megjegyzés. Ha egy passzív rekord *id*-je -3, akkor ez egy *autoConnBox* passzív rekordja, amelyre csakis -2 azonosítójú (*connBox ancestorThread*) aktív rekordból szabad hivatkozni. Ha egy passzív rekord *id*-je -2, akkor ez egy olyan passzív rekord, amelyre tilos aktívnak hivatkoznia. Ezért akkor megfelelő az ellenőrzés eredménye, ha a hivatkozó rekordok halmaza üres. Ha az id -1, akkor ez egy *auto* csatorna, amelyre -1 azonosítójú aktív rekordból szabad hivatkozni (*startGraph* vagy *connBox thisThread*). Amennyiben a passzív csatorna fix (adott) id-vel rendelkezik ($P.id \geqq 0$), úgy a párjának is fix id-vel kell rendelkeznie. Ha a passzív csatorna nem fix azonosítójú, úgy a párja sem lehet az. Ezen felül minden esetben vizsgálni kell a típusegyezőséget is.

2.66. definíció (PROJEKT P→A ELLENŐRZŐ). Egy $fp_{chkPA}\colon \langle R_p\rangle \times \langle R_a\rangle \to \mathbb{B}$, $fp_{chkPA}(P, A) = B$ függvényt projekt szintű *passzív-aktív* csatorna-kapcsolat ellenőrző függvénynek nevezzük, ahol $P = \langle p_0, p_1, \ldots, p_{n-1}\rangle$ esetén $B \coloneqq f_{chkPA}(p_0, A) \wedge \wedge f_{chkPA}(p_1, A) \wedge \cdots \wedge f_{chkPA}(p_{n-1}, A)$.

2.68. megjegyzés. Ezen függvény minden egyes passzív rekordra ellenőrzi a rá hivatkozó aktív rekordokat. Akkor kapunk végeredményül *true* (helyes) eredményt, ha minden egyes passzívra hivatkozás rendben van.

2.6.8. Aktív → passzív ellenőrző függvények

Az aktív→ passzív ellenőrző függvények is a párkeresés eredményét értékelik ki az aktív oldal szempontjából vizsgálva. A párkeresés helyes, ha minden hivatkozás egyszeri, megfelelő jellegű és típushelyes. Az ellenőrzés gondolatmenetét a 2.44. megjegyzés, illetve a 2.69. megjegyzés tartalmazza.

2.67. definíció (A→P ELLENŐRZŐ). Az $f_{chkAP}\colon R_A \times \langle R_p\rangle \to \mathbb{B}$, $f_{chkAP}(A, S) = = B$ függvényt *aktív-passzív csatornakapcsolat-ellenőrző* függvénynek nevezzük, ahol $T \coloneqq fap_{id}(A, S) \cup fap_{order}(A, S), T = \langle p_0, p_1, \ldots p_{n-1}\rangle$ esetén

- ha $A.id \geqq 0$ akkor ha $sizeof(T) = 1 \wedge A.id = p_0.id \wedge p_0.type = A.type$ akkor $B \coloneqq true$
- ha $A.id = -1$ akkor ha $sizeof(T) = 1 \wedge p_0.id = -1 \wedge p_0.type = A.type$ akkor $B \coloneqq true$
- ha $A.id = -2$ akkor ha $sizeof(T) = 1 \wedge p_0.id = -3 \wedge p_0.type = A.type$ akkor $B \coloneqq true$
- minden más esetben $B \coloneqq false$.

2.69. megjegyzés. Egy aktív rekordnak mindig pontosan egy passzív rekordra kell mutatni, amelynek a típusa is egyezik vele. Ha az aktív rekord fix azonosítójú ($A.id \geqq$

0), akkor a passzív pár is fix azonosítójú kell legyen. Ha az aktív rekord nem fix azonosítójú, akkor a passzív pár sem lehet az. A -1 azonosítójú aktív rekordhoz -1 azonosítójú passzív rekord kell tartozzon (előbbit *startGraph* vagy *connBox thisThread*, utóbbit *auto* csatorna esetén kapjuk). A -2 azonosítójú aktív rekord a *connBox ancestorThread*, mely csak *autoConnBox*-hoz csatlakozhat.

2.68. definíció (PROJEKT A→P ELLENŐRZŐ). Legyen $fp_{chkAP}\colon \langle R_a\rangle \times \langle R_p\rangle \to \mathbb{B}$, $fp_{chkAP}(A,P) = B$. Ezt a függvényt projekt szintű *aktív-passzív* csatornakapcsolat-ellenőrző függvénynek nevezzük, ahol $A = \langle a_0, a_1, \ldots, a_{n-1}\rangle$ esetén $B \coloneqq f_{chkAP}(a_0,P) \wedge \wedge f_{chkAP}(a_1,P) \wedge \cdots \wedge f_{chkAP}(a_{n-1},P)$.

2.70. megjegyzés. Ezen függvény minden egyes aktív rekordra ellenőrzi az általa hivatkozott passzív rekordokat. Akkor kapunk végeredményül *true* (helyes) eredményt, ha minden egyes aktív passzívra hivatkozása rendben van.

2.6.9. Csatornapárok statikus helyességének definiálása

A korábban ismertetett ellenőrző függvények helyes alkalmazását definiáljuk az alábbiakban. A 2.69. definíció összesíti a korábban bemutatott ellenőrzési módszerek alkalmazását a statikus helyesség megállapításához.

2.69. definíció (CSATORNÁK SZERINT HELYES). Valamely $D_p \in \mathbb{D}$ D-Box projekt, $D_p = \langle D_0, D_1, \ldots, D_{n-1}\rangle$ csatornái *statikusan helyes párokat alkotnak*, amennyiben $P_{pi} = fp_{PI}(D_p), A_{pi} = fa_{PI}(D_p), P_{po} = fp_{PO}(D_p), A_{po} = fa_{PO}(D_p)$, esetén az

$$\Big(fp_{chkPA}(P_{pi},A_{po}) \wedge fp_{chkPA}(P_{po},A_{pi})\Big) \wedge \Big(fp_{chkAP}(A_{pi},P_{po}) \wedge fp_{chkAP}(A_{po},P_{pi})\Big)$$

kifejezés értéke *true*.

2.71. megjegyzés. A fenti ellenőrző függvény a 2.43. megjegyzésben megadott mind a négy ellenőrzést tartalmazza. A hivatkozásoknak helyesnek kell lenni az

- input passzív → output aktív
- output passzív → input aktív
- input aktív → output passzív
- output aktív → input passzív

ellenőrzési irányok mindegyikében.

2.7. Algráf ellenőrzési problémák

A projektbeli D-Box dobozok részhalmazokba sorolhatók. A halmazba sorolásukat a halmaz sorszámával (algráf azonosító) adjuk meg. Az azonos halmazbeli dobozok jellemzően egymással kommunikálnak, ezért önállóan működő, zárt, irányított gráfot alkotnak. Az algráf (részgráf) fogalom jelen esetben arra utal, hogy a teljes projekt futás

közbeni alakja egy irányított gráfot ír le, melyben az azonos halmazbe eső dobozok összefüggő részgráfot fednek le.

Az algráf futás közben indított példánya a *szál*. Egyetlen statikus algráfot dinamikusan többször is el lehet indítani, így ugyanazon algráf több példányban is létezhet. Ezek egymástól a szálazonosítókban különböznek. Az algráfok egymással közvetlenül nem kommunikálhatnak, csak a szülő (indító) szál, és a gyermek (indított) szál kommunikálhat egymással. A szülő szálra a gyerek szálban az *ancestorThread* azonosítóval lehet hivatkozni. Valamely szál saját magára a *thisThread* azonosítóval hivatkozhat.

A korábban definiált aktív-passzív párosító rekordok segítségével igazolhatjuk, hogy a projektet egyetlen hatalmas gráfként szemlélve a csatornák párban vannak-e, a dobozok közötti adatáramlás biztosított-e, nincsenek olyan csatornák, amelyekre csak fogyasztó vagy csak termelő doboz kapcsolódik.

Ugyanakkor a fenti ellenőrzés során csakis a doboz azonosítókat és a csatorna azonosítókat vettük figyelembe. Amennyiben egy csatorna két vége más-más algráfba tartozó dobozhoz tartozik, úgy a helyes működés mégsem feltétlenül biztosított, mivel elképzelhető olyan szituáció, hogy valamely algráf még nem került indításra, ekkor a termelő és fogyasztó oldal létezése futás közben mégsem áll elő. Az aktív-passzív párok létezése és megfelelősége szükséges, de nem elégséges feltétel a projekt futás közbeni helyes viselkedéséhez. Ehhez figyelembe kell venni az egyes algráfok futás közbeni viselkedését is.

A projekt indítása során a futtató rendszer az 1-es algráf dobozait fogja betölteni, és indítani. Ezek további algráfokat tölthetnek be dinamikusan, de ez már futás közbeni folyamat. Ha egy projekt nem tartalmaz egyetlen `1`-es algráfba eső dobozt sem, úgy a projekt indítása gyakorlatilag nem megvalósítható.

Az 1-es algráfból őt indító gráfra hivatkozni (*ancestorThread*) nem lehet, őt a futtató rendszer indítja. 1-es algráfot futás közben indítani *startGraph*-fal nem lehet, csak eltérő sorszámú algráfot.

Zártság: amennyiben valamely A algráf valamely további B algráfot (szál) indít, úgy a B számára az A aktuális példánya lesz a szülő szál (*ancestorThread*). A B-ben definiált csatornák csakis a B-beli doboz felé irányulhatnak, kivéve a *connBox ancestorThread ...* kifejezést, amelynek A-beli doboz felé kell mutatnia. Az A-ban megadott, B-t indító *startGraph* kifejezésbeli dobozazonosító valamely B-beli dobozt kell azonosítsa. Ha minden egyes algráfbeli csatornák csakis saját magán belülre irányulnak, vagy legfeljebb a szülő szál felé, vagy az általa indított további algráfok felé irányulnak, akkor az algráf zártnak tekinthető. Ha minden egyes algráf zárt, akkor a teljes projekt is zárt, vagyis a kommunikációs csatornák kezelése helyes.

Külön figyelnünk kell az algráfban esetleg előforduló fix azonosítójú csatornákra. Mivel egy algráfból több példány is indítható, ez esetben már a fix azonosítójú csatornák több példányáról lehetne beszélni, amely nem megengedett (futás közben a csatorna

azonosítók a teljes projektre nézve egyedieknek kell lenniük). Logikus észrevétel: fix azonosítójú csatorna csakis az 1-es algráfban fordulhat elő.

Amennyiben egy *startGraph*-fal indított algráf elindul, úgy a feldolgozandó adatokat a doboz fogja átadni, amelyik azt *startGraph*-fal elindította. A továbbiakban két dolog történhet:

- az algráf belépő doboza az adatokat fogadja, saját algráfbeli dobozok segítségével feldolgozza, és az eredményt kimenő csatornákra helyezi, ahonnan az őt indító „szál" el tudja olvasni,
- a feldolgozás zárt, az algráf a feldolgozott eredményt önállóan menti diszkre, vagy más módon kezeli annak végleges sorsát. Kimenő adatai nincsenek.

Az első esetben az algráf a hagyományos programozási nyelvekben ismert *függvény*szerű szerepet tölt be, a második esetben *eljárás*szerű szerepet.

Amennyiben bizonyos dobozok olyan algráfba vannak sorolva, amelyekre egyetlen *startGraph* kifejezés sem hivatkozik, úgy ezen dobozok indítására futás közben nem kerülhet sor. Ez ugyanaz az eset, mintha egy programkódban olyan függvény szerepelne, amelyet végül is nem használunk fel. Ez elvileg szintaktikai hibát nem okoz, de figyelmeztető üzenet kiírására érdemes. Ezért ezt is ellenőrizni kell.

2.7.1. Előkészületek a helyesség megfogalmazásához

Az alábbiakban olyan segédfüggvényeket adunk meg, melyekre a későbbiekben hivatkozni fogunk.

2.70. definíció (Algráfot alkot). Egy $D_p \in \mathbb{D}, D_{p'} \subseteq D_p$ halmaz *algráfot alkot*, amennyiben $D_{p'} \neq \emptyset$, $D_{p'} = \langle D_0, D_1, \ldots, D_{n-1} \rangle$ esetén $\forall D_a, D_b \in D_{p'}$ dobozokra $D_a.subGraphID = D_b.subGraphID$.

2.71. definíció (Algráf azonosító). Az $f_{gid}\colon \mathbb{D} \to \mathbb{N}$, $f_{gid}(D_p) = n$ függvény értékét a D_p algráf azonosítójának nevezzük, ahol ha $D_p \neq \emptyset$, $D_p = \langle D_0, D_1, \ldots, D_{n-1} \rangle$, és ha $\nexists D_a, D_b \in D_p$, hogy $D_a.subGraphID \neq D_b.subGraphID$, akkor $n \coloneqq D_0.subGraphID$, különben $n \coloneqq -1$.

2.72. megjegyzés. Ha minden doboz ugyanazon algráf azonosítóval rendelkezik, akkor az f_{gid} függvény ezen értéket határozza meg, különben -1-et.

2.72. definíció (Algráf kikeresŐ). Legyen $f_{sup}\colon \mathbb{D} \times \mathbb{N} \to \mathbb{D}$, $f_{sup}(D_p, n) = D_a$ függvény, ahol $D_a \coloneqq \{D\colon D \in D_p, D.subGraphID = n\}$.

2.73. megjegyzés. Az f_{sup} függvény kikeresi egy adott algráf azonosítóhoz tartozó összes dobozt.

2.73. definíció (startGraph kereső). Az $f_{sg}\colon D_p \to \mathcal{P}(D_{startGraph})$, $f_{sg}(D_p) = r$ legyen olyan függvény, ahol ha $D_p = \langle D_0, D_1, \ldots, D_{n-1}\rangle$, akkor

$$r \coloneqq \{d\colon \exists D \in D_p, d \in D.OutProt.OChannels \wedge d \simeq D_{startGraph}\}.$$

2.74. megjegyzés. Az f_{sg} függvény projekt szinten kikeresi az összes *startGraph* kifejezést.

2.74. definíció (connBox kereső). Az $f_{cb}\colon \mathbb{D}\times\mathbb{H} \to \mathcal{P}(D_{connBox})$, $f_{cb}(D_p, thr) = r$ legyen olyan függvény, ahol ha $D_p = \langle D_0, D_1, \ldots D_{n-1}\rangle$, akkor

$$r \coloneqq \{d\colon \exists D \in D_p, d \in D.OutProt.OChannels \wedge d \simeq D_{connBox} \wedge d.thr = thr\}.$$

2.75. megjegyzés. Az f_{cb} függvény projekt szinten kikeresi az összes, adott szálra vonatkozó *connBox* kifejezést.

2.75. definíció (connBox dobozok). Legyen az $f_{cbd}\colon \mathbb{D}\times\mathbb{H} \to \mathbb{D}$, $f_{cbd}(D_p, thr) = r$ olyan függvény, ahol ha $D_p = \langle D_0, D_1, \ldots, D_{n-1}\rangle$, akkor

$$r \coloneqq \{D\colon D \in D_p, \exists d \in D.OutProt.OChannels \wedge d \simeq D_{connBox} \wedge d.thr = thr\}.$$

2.76. megjegyzés. Az f_{cbd} függvény hasonló az f_{cb} függvényhez, csak ez nem a *connBox* kifejezéseket keresi ki, hanem a *connBox* kifejezéseket tartalmazó dobozokat.

2.76. definíció (jó projekt). Valamely $D_p \in \mathbb{D}$ D-Box projektet *jó projektnek* nevezünk, ha $D_p = \langle D_0, D_1, \ldots, D_{n-1}\rangle$ esetén $\nexists D_a, D_b \in D_p$, $D_a \neq D_b$, hogy $D_a.boxid = D_b.boxid$.

2.77. megjegyzés. Egy projektet ***jó***nak tekintünk, ha a projektben szereplő minden doboznak különböző az azonosítója.

2.77. definíció (box kereső). Az $f_D\colon \mathbb{D} \times String \to \mathbb{D}$, $f_D(D_p, BoxID) = H_D$ legyen olyan függvény, ahol ha $D_p = \langle D_0, D_1, \ldots, D_{n-1}\rangle$, akkor $H' \coloneqq \{D\colon D \in D_p, D.BoxID = BoxID\}$.

2.78. megjegyzés. Az f_D függvény egy adott D-Box azonosítóhoz tartozó D-Box definíciókat keresi ki a projektből. Elvileg egy ilyen doboz azonosító egyedi, ezért ez a függvény egy elemű halmazt ad meg találat esetén. Nem létező azonosító esetén üres halmazt ad meg. Ezen függvényt jól tudjuk majd használni pl. annak ellenőrzésére, hogy egy *connBox*ban szereplő doboz azonosító egyáltalán létezik-e a projektben.

2.7.2. Projekt indíthatósági probléma

Az algráfokkal kapcsolatos első vizsgálódás a projekt indíthatóságának kérdése. A projekt indítása az 1-es algráfba tartozó dobozok indítását jelenti. Amennyiben nincs egyetlen ilyen doboz sem, a projekt nem indítható.

2.78. definíció (ALGRÁF AZONOSÍTÓK HALMAZA). Legyen az $f_{sgid}\colon \mathbb{D} \to \mathcal{P}(\mathbb{N})$, $f_{sgid}(D_p) = r$ olyan függvény, ahol ha $D_p = \langle D_0, D_1, \dots D_{n-1}\rangle$, akkor

$$r \coloneqq \{id\colon \exists D \in D_p, id = D.subGraphID\}.$$

2.79. megjegyzés. Az f_{sgid} függvény meghatározza a projektben előforduló gráfazonosítók halmazát.

2.79. definíció (PROJEKT INDÍTHATÓ). Egy $D_p \in \mathbb{D}$ projektről azt mondjuk, hogy *indítható*, ha $1 \in f_{sgid}(D_p)$.

2.7.3. Algráf indítási probléma

Amennyiben egy *startGraph* valamely algráfra hivatkozik (indítani szeretné), úgy az ugyanebben a *startGraph* kifejezésben szereplő hivatkozott doboznak is ebbe az algráfba kell tartoznia.

2.80. definíció (DINAMIKUS INDÍTÁS RENDBEN). Azt mondjuk, hogy egy $D_p \in \mathbb{D}$ jó projekt *dinamikus indításai rendben vannak*, ha a projektben szereplő $\forall d \in f_{sg}(D_p)$ startGraph csatorna leíró esetén a $H \coloneqq f_D(D_p, d.BoxID)$ halmazra teljesül, hogy $sizeof(H) = 1$, $H = \{D_0\}$, $d.sid = D_0.subGraphID$ és $d.sid \neq 1$.

2.80. megjegyzés. Vizsgáljuk, hogy minden *startGraph* kifejezésben szereplő gráfazonosító szerepel-e a projektben, és a *startGraph*-ban hivatkozott doboz is a megadott algráfba tartozik-e. Ezen algráf nem lehet az 1-es algráf.

2.81. definíció (ALGRÁFOT INDÍT). Legyen $D_p \in \mathbb{D}, D_a, D_b \subset D_p$ két algráf. Azt mondjuk, hogy D_a gráf a D_b gráfot *indítja*, ha $\exists d \in f_{sg}(D_a)$, hogy $d.sid = f_{gid}(D_b)$.

2.81. megjegyzés. Egy D_a indít egy D_b algráfot, ha a D_a valamely dobozában van olyan *startGraph* kifejezés, amely a D_b algráf azonosítóját tartalmazza.

2.7.4. Algráf kimenő csatornák vizsgálata

Valamely algráf dobozaiban definiált „kifelé mutató” (másik algráf példányba irányuló) csatornák esetén vizsgálandó a hivatkozási helyesség. Csak a szülő gráfba visszafele, vagy általa indított gyerek algráf dobozaiba irányulhatnak ezek a hivatkozások.

2.82. definíció (FIX CSATORNÁKTÓL MENTES). Legyen $D_p \in \mathbb{D}, D_a \subset D_p$ algráfot *fix csatornáktól mentesnek* nevezzük, ha a D_a algráfot alkotó D-Box dobozok egyikének sem input, sem output csatornái között sem szerepel D_{fix} csatorna leíró.

A kimenő csatornák csakis az alábbi típusúak lehetnek:

- *null*: mely esetben nincs kimenő csatorna,
- *fix*: ilyen egy nem 1-es algráfban nem fordulhat elő,
- *connBox thisThread*: mely ugyanezen gráf példányon (thread) belülre irányul,
- *connBox ancestorThread*: mely a szülő példány (thread) felé irányul,
- *startGraph*: mely gyerek algráfot indít, így biztosan megfelelő a kommunikációja.

Az algráf kifelé irányuló zártságát csak a *connBox ancestorThread* esetén kell vizsgálni, mivel a többi lehetőség nem okozhatja a zártság megtörését.

2.83. definíció (VISSZAFELÉ ZÁRT). Legyen $D_p \in \mathbb{D}$ egy *jó* projekt, $n \in \mathbb{N}$, $D_a = f_{sup}(D_p, n)$ algráf. $H \coloneqq f_{cb}(D_a, ancestorThread) = \{d_0, d_1, \ldots, d_{n-1}\}$ ezen D_a algráfban szereplő *connBox* kifejezések. Azt mondjuk, hogy a D_a *algráf visszafelé zárt*, ha $\forall i, j \in [0, n-1]$ esetén $D_i.subGraphID = D_j.subGraphID$ (ahol $A \coloneqq f_D(D_p, d_i.boxid)$, $B \coloneqq f_D(D_p, d_j.boxid)$ a hivatkozott dobozok, $sizeof(A) = 1$, $sizeof(B) = 1$, $A = \{D_i\}$, $B = \{D_j\}$).

2.82. megjegyzés. Minden D_b gráfbeli *connBox „ancestorThread"* kifejezésben szereplő doboz ugyanazon algráfba tartozik. Vegyük észre azonban, hogy jelen definíció nem tárgyalja, hogy a D_a algráf által ezen hivatkozott „visszafelé" algráf valóban „szülő" algráfja-e vagy sem.

2.84. definíció (CSAK SZÜLŐJÉVEL KOMMUNIKÁL). Legyen $D_p \in \mathbb{D}$ *jó* projekt, $m \in f_{sgid}(D_p)$ projektbeli algráf azonosító, $D_a = f_{sup}(D_p, m)$ az m által azonosított algráf, $H \coloneqq f_{cb}(D_a, ancestorThread) = \{d_0, d_1, \ldots, d_{n-1}\}$. Azt mondjuk, hogy a D_a *algráf csak a szülőjével kommunikál*, ha $n = 0$, vagy $n > 0$ esetén $D \coloneqq f_D(D_p, d_0.boxid)$ indító dobozt tartalmazó $D_b \coloneqq f_{sup}(D_p, D.subGraphID)$ algráf a D_a algráfot *indítja*.

A 2.84. definícióbeli zártság csak azt tárgyalja, hogy az *ancestorThread* kimenetek minden esetben valóban a szülő gráf felé irányulnak-e. Ez a teljes szintaktikai helyességhez még nem elegendő. Előállhat az a helyzet, hogy egy D_b gráf több pontjából is indítjuk (*startGraph*-fal) valamely D_a gráfot, mely esetekben mindig a D_b a szülő gráf, de ez kevés. A *connBox*ok ugyanis (elvileg) kétfajta input csatornára tudnak csatlakozni:

- *auto* input csatornára. Az ilyen input csatornák csak egyetlen példányban léteznek a dobozukban, vagyis ez esetben a D_a algráfot csak egyetlen helyről, egyetlen példányban szabad elindítani a D_b gráf által.

- *autoConnBox boxid*, amely input csatornák lekérik az adott „boxid" azonosítójú doboz hány példányban indította el a D_a gráfot, és ennyi példányban léteznek majd futás közben.

Mivel nem indokolt az *auto* csatornákra csatlakozás az algráfból, ezért ezt egyszerűen megtiltjuk. Ezt egyébként az aktív-passzív párkereső függvények ki is szűrik, ezért annak ellenőrzése, hogy minden érintett csatorna a szülő oldalon *autoConnBox* típusú, nem szükséges.

Ha egy D_a gráf ilyen *autoConnBox* input csatornákra kíván visszafelé csatlakozni, akkor ezen *autoConnBox*ok attól a doboztól kell lekérjék az indított algráfok számát, amelyik ténylegesen indította őket. Eközben nem azonosítják, melyik algráfról is van szó, ezért egy ilyen indító doboz csak egy *startGraph* kifejezést tartalmazhat. Amennyiben az indított algráf *connBox ancestorThread*-del visszacsatlakozik a szülő szálra (függvény-szerű viselkedés), úgy ezt az algráfot csakis egyetlen pontról szabad indítani a szülő gráfban (és ezen indító doboz más algráfot nem indíthat).

2.85. definíció (connBox output párok). Legyen $D_p \in \mathbb{D}$ jó projekt, $m \in$ $\in f_{sgid}(D_p)$, $D_a \coloneqq f_{sup}(D_p, m)$ algráf, $H \coloneqq f_{cb}(D_a, ancestorThread)$ azok a *connBox* kifejezések, amelyek ezen algráfból a szülő felé irányulnak, $H = \{d_0, d_1, \ldots, d_{n-1}\}$. $\forall i \in [0, n-1]$ esetén jelöljök D_i-val a d_i *connBox* által hivatkozott dobozt ($X_i =$ $= f_D(D_p, d_i.boxid)$, $sizeof(X_i) = 1, X = \{D_i\}$). Legyen $P_i \coloneqq fp_D I(D_i)$ halmaz ezen doboz input passzív csatorna ellenőrző rekordjai, $P_i' \subset P_i$ halmazzal jelöljük azokat a rekordokat, melyek az $fp_{connBox}(d_i)$ által kerültek generálásra $P_i' = \{p_0, p_1, \ldots, p_{k-1}\}$. Ezen P_i' halmaz a d_i connBox kifejezés aktív rekordjainak passzív párjai. A $P_{sup} =$ $= \{P_0', P_1', \ldots, P_{n-1}'\}$ halmazt a D_a *algráfhoz tartozó* autoConnBox *passzív rekordoknak* nevezzük.

2.83. megjegyzés. A P_{sup} az algráfban (gyerek gráf) szereplő *connBox* output csatornák input párjait azonosítja. A P_{sup} elemei a szülő gráfban szereplő *autoConnBox* csatornák passzív ellenőrző rekordjai. Ha nincsenek *connBox* output csatornák (eljárásszerű viselkedés), akkor ezen halmaz értelemszerűen üres.

2.86. definíció (startGraph hivatkozott dobozok). Legyen $D_p \in \mathbb{D}$ jó projekt, $m \in f_{sgid}(D_p)$, $D_a \coloneqq f_{sup}(D_p, m)$ algráfhoz tartozó *autoConnBox* passzív rekordok halmaza, $P_{sup} = \{p_0, p_1, \ldots, p_{n-1}\}$. Legyen $D_{sg} \coloneqq \{D \colon D = f_D(D_p, p.boxid) \forall p \in P_{sup}\}$ halmaz a D_a gráfhoz tartozó *hivatkozott startGraph dobozok* halmaza.

2.84. megjegyzés. A D_{sg} halmazban azok a dobozok vannak, amelyekre a szülő gráf *autoConnBox* kifejezései utalnak. Az *autoConnBox* csatorna leíró kifejezések *boxid*-ket tartalmaznak, melyek azon dobozokat azonosítják, amelyek a D_a algráfot indítják.

2.87. definíció (szülőjével megfelelően kommunikál). Legyen $D_p \in \mathbb{D}$ jó projekt, $m \in f_{sgid}(D_p), m > 1, D_a = f_{sup}(D_p, m)$ valamely algráf. Legyen a D_a algráfhoz tartozó hivatkozott *startGraph* dobozok halmaza D_{sg}.

- ha $sizeof(D_{sg}) = 0$ úgy a D_a algráf *a szülőjével megfelelően kommunikál.*
- ha $sizeof(D_{sg}) = 1$ ($D_{sg} = \{D_0\}$) úgy jelölje $S \coloneqq f_{sg}(\langle D_0 \rangle)$. Ha $sizeof(S) = 1$ ($S = \{d_0\}$), és teljesül $d_0.sid = m$, úgy szintén mondhatjuk, hogy a D_a algráf *a szülőjével megfelelően kommunikál.*

2.85. megjegyzés. Vagyis ha nincs a D_a algráfnak szülő felé mutató csatornája, akkor triviális, hogy a szülőjével megfelelően kommunikál. Ha van kimenő csatornája, akkor keressük ki azokat a dobozokat, akik ezen D_a algráfot indítják (D_{sg} halmaz). Ezen halmaz ekkor csak egyetlen elemű lehet. Vegyük ezen egyetlen doboz (D_0) *startGraph* kifejezéseit (S halmaz). Ezen halmaz egyetlen elemű kell legyen, ezen dobozban csak egyetlen *stargGraph* lehet. Vegyük ezen egyetlen *startGraph* kifejezést (d_0), és vizsgáljuk meg, hogy ez a *startGraph* valóban a D_a algráfot indítja-e. Ha igen, akkor a D_a kimenő kommunikációja szintén helyes, lehet abban bízni, hogy a kimenő csatornák olyan *connBox* csatornákra csatlakoznak fel, amelyek megfelelően sok kollekcióba tartoznak majd.

2.86. megjegyzés. A fenti *megfelelő kommunikáció* automatikusan teljesül, ha az algráfnak nincs kimenő csatornája. Egy ilyen eljárásszerű algráfot tetszőlegesen sok más algráf tetszőlegesen sok dobozából (akár egyetlen dobozon belül többször is) el szabad indítani a *startGraph* segítségével.

2.7.5. Algráfok helyességének vizsgálata

Az algráfok együttműködésével kapcsolatos vizsgálódás a *helyesen zártság* fogalmának kialakításával zárul. Ezen definíció összefoglalja az eddigi eredményeket.

2.88. definíció (algráfok helyesen zártak). Egy $D_p \in \mathbb{D}$ *jó* projekt *algráfjai helyesen zártak*, ha $\forall n \in f_{sgid}(D_p)$, $D_n \coloneqq f_{sup}(D_p, n)$ algráf esetén teljesülnek az alábbi feltételek:

- $n > 1$ esetén D_n fix csatornáktól mentes (2.82. definíció),
- D_n visszafelé zárt (2.83. definíció),
- D_n csak a szülőjével kommunikál(2.84. definíció),
- D_n a szülőjével megfelelően kommunikál (2.87. definíció).

2.87. megjegyzés. Mint említettük, fix csatorna csak az 1-es algráfban fordulhat elő. A visszafelé zártság teljesül, ha *connBox ancestorThread* által hivatkozott doboz valóban ugyanabba az algráfba tartozik. A csak szülővel kommunikál, ha a *connBox ancestorThread* dobozok valóban a szülő algráfbeliek. A megfelelő kommunikáció során

pedig azt ellenőrízzük le, hogy a *connBox ancestorThread* dobozok valóban az algráfot indító *startGraph* kifejezéstől kérik-e le az indított példányszámot.

2.8. A D-Box projekt statikus szemantikai helyessége

Az eddigi eredmények összefoglalásaképp a statikus szemantikai helyesség feltételeit összesítjük.

2.89. definíció (SZEMANTIKAILAG HELYES). Egy $D_p \in \mathbb{D}$ D-Box projekt *statikus szemantikailag helyes*, ha

- csatornái statikusan helyes párokat alkotnak (2.69. definíció),
- dinamikus indításai rendben vannak (2.80. definíció),
- algráfjai helyesen zártak (2.88. definíció).

2.88. megjegyzés. A *statikus szemantikai helyesség* magában foglalja, hogy a projektben szereplő csatornák mindegyike helyesen van párosítva, és a dinamikus algráf indítások helyesek, és az algráfból szülő felé irányuló output csatornák dinamikusan is megfelelően fognak csatlakozni.

Ezzel befejeztük a D-Box nyelv statikus szemantikai helyességének tárgyalását.

2.9. Összefoglalás

Ebben a fejezetben formálisan megadtuk a D-Box nyelvi leírások statikus szemantikai szabályait. A vizsgálatok során elemeztük a dobozdefiníciókban szereplő csatornák helyes használatát, különös figyelemmel arra, hogy minden csatornát pontosan egy doboz használjon input, és pontosan egy doboz output csatornaként. Vizsgáltuk a projekt indíthatóságát, valamint dinamikus indításainak helyességét. A projekt indíthatóságát az 1-es algráfbeli dobozok létezése mutatja, míg a dinamikus algráfok indíthatóságánál ügyelni kell a hivatkozott algráf és hivatkozott doboz összetartozására. Az algráfok zárt kommunikációja szükséges, hogy a dinamikusan indított és azonosított csatornák esetén is teljesüljön a pontosan egy input és output célú felhasználás.

3. fejezet

A D-Box nyelv kommunikációs primitívjeinek specifikációja

Az alábbiakban megadjuk a D-Box nyelvi elemek, csatornák, protokollok, a csatornák indításának, az algráfok indításának specifikációját.

3.1. Csatorna vezérlő szimbólumok bemutatása

A csatorna a *D-Box* projekt egyik alapvető eleme. A csatornákon keresztül tudnak a dobozok egymással kommunikálni. A csatornákat a futtató rendszer indítja el, amennyiben valamely doboz indítási kérelmet intéz felé. A kérés során meg kell jelölni az indítandó csatorna típusát.

Egy csatorna a saját azonosítóján felül egy puffert tartalmaz, melybe véges mennyiségű adatelemet képes átmenetileg tárolni. Ezen puffer az író és az olvasó doboz eltérő működési sebességét képes bizonyos mértékben kompenzálni.

3.1. jelölés (CSATORNA VEZÉRLŐ SZIMBÓLUM). Vezessük be a következő jelöléseket:

- *EoS* az allista vége (*End of Sublist*),
- *EoL* a lista vége (*End of List*),
- *EoC* a csatorna vége (*End of Channel*).

Ekkor a $V_{csv} \coloneqq \{EoS, EoL, EoC\}$ halmazt *csatornavezérlő szimbólumok* halmazának nevezzük.

3.1. definíció (CSATORNA PUFFER TÍPUS). A $T_{cs} \coloneqq T_\tau \cup V_{cs}$ halmazt *csatornapuffer típusnak* nevezzük.

3.2. definíció (CSATORNA MŰVELETI TÍPUS). A $T_{cs'} \coloneqq T_{cs} \cup \{undefined\}$ halmazt *csatorna műveleti típusnak* nevezzük.

3.1. megjegyzés. Mint később látni fogjuk, a csatorna *kiolvas* művelete adhat vissza normál adatelemet (τ), valamely csatornavezérlő szimbólumot (T_{cs}), illetve hiba esetén egy speciális *undefined* értéket is.

3.2. jelölés (HIBAKÓD). Az $\mathbb{E} \coloneqq \{e_{ok}, e_{err}, e_{finished}\}$ halmazt *csatorna hibakód halmaznak* nevezzük.

3.2. megjegyzés. A *csatorna hibakódok* a csatorna műveleteknél lesz majd fontos, az e_{ok} a művelet sikerességét, az e_{err} a művelet sikertelenségét jelöli. Az $e_{finished}$ a csatornára írás során kerül majd felhasználásra.

3.2. Csatornaműveletek specifikációja

A futtató rendszer által indított csatornákkal kapcsolatosan négy fő műveletről beszélhetünk. Ezen műveletek során elsősorban a csatorna pufferének tartalma módosul. A csatornát legfeljebb két doboz használja, de akár egyidőben is kezdeményezhetik a saját műveletük végrehajtását. A csatorna a műveleteit atomi szinten kezeli, a kizárólagos hozzáférést saját hatáskörben egy *monitor* segítségével oldja meg.

3.3. definíció (CSATORNA). Egy $C(uid, T, max, E)$ formális négyest *csatornának* nevezünk, ahol

- $uid \in \mathbb{N}$ a csatorna azonosító,
- $T \in T_\tau$ a csatorna alaptípusa,
- $max \in \mathbb{N}_+$ pozitív szám, a puffer maximális tárolókapacitása,
- $E \in \langle T \cup V_{cs} \rangle$, $E = \langle e_0, e_1, \ldots, e_{n-1} \rangle$ a csatorna pufferében tárolt véges elemsorozat, $0 \leqq sizeof(E) \leqq max$.

3.3. megjegyzés. A csatorna alaptípusa ténylegesen valamilyen átvihető típus ($T \in \in T_\tau$), de a pufferben az ilyen típusú elemeken túl csatornavezérlő szimbólumok is tárolódhatnak.

3.4. megjegyzés. Egy $C(uid, T, max, E)$ csatornán az alábbi műveleteket értelmezzük:

- *init()*, a csatorna alaphelyzetbe állítása,
- *shutDown()*, a csatorna kikapcsolás,
- *store()*, a csatornapufferbe elem elhelyezése,
- *retrieve()*, a csatornapufferből elem kiolvasása.

3.5. megjegyzés. A csatorna a termelő-fogyasztó problémában [41] bemutatott puffer viselkedésű. Rendelkezik kizárólagos hozzáférést biztosító monitorral [42], mely biztosítja a fenti négy művelet atomicitását. A műveletek mindegyike kizárólagos hozzáférés mellett fut, vagyis egyidőben csak egy művelet lehet aktív.

3.4. definíció (INIT MŰVELET). Az *init*: $C(uid, T, max, E) \times \mathbb{N} \times T_\tau \times \mathbb{N} \to \mathbb{E} \times \times C(uid, T, max, E)$, $init(cs, uid, t, n) = (e', cs')$ leképezést *csatorna init() műveletnek* nevezzük, ahol $cs' \coloneqq C(uid, t, n, \emptyset)$, $e' \coloneqq e_{ok}$.

3.6. megjegyzés. Az inicializálás során a csatorna eltárolja az azonosítóját (egyedi sorszám), a típust, a tárolt sorozat egyenlőre üres, a maximális elemszám definiált lesz. Az *init()* művelet mindig sikeres.

3.5. definíció (SHUTDOWN MŰVELET). A *shutDown*: $C(uid, T, max, E) \to \mathbb{E} \times \times C(uid, T, max, E)$, $shutDown(cs) = (e', cs')$ leképezést *csatorna shutDown műveletnek* nevezzük, ha $cs' \coloneqq C(cs.uid, cs.T, 0, \emptyset)$, a puffer kizárólagos hozzáférését birtokló és a hozzáférésre várakozó processzek mindegyike sikertelen végrehajtást tapasztal. A puffer a továbbiakban semmilyen műveletetet nem fogad: minden kezdeményezett művelete azonnal sikertelen végrehajtást eredményez. Ez utóbbi alól kivételt képez az újból végrehajtott *init* művelet, mely újra inicializálja a csatornát, s mely után a csatorna újra képes műveleteket végezni.

3.6. definíció (STORE MŰVELET). A *store*: $C(uid, T, max, E) \times T \cup V_{cs} \to \mathbb{E} \times \times C(uid, T, max, E)$, $store(cs, x) = (e', cs')$ leképezést *store műveletnek* nevezzük, ahol

- végrehajtunk egy termelő-fogyasztó problémában ismert *add()* lépéssorozatot. Amennyiben ez a művelet sikeresen befejeződik, úgy $e' \coloneqq e_{ok}$, $cs' \coloneqq C(cs.uid, cs.T, cs.max, cs.E \oplus x)$,
- különben $cs' \coloneqq cs$, $e \coloneqq e_{err}$.

3.7. megjegyzés. A termelő-fogyasztó *add()* művelet során a termelő elsősorban megszerzi a kizárólagos hozzáférést a pufferhez. Amennyiben az tele van, úgy várakozni kezd mindaddig, míg a pufferben egy üres hely nem keletkezik. Ezen várakozásnak jelen esetben nincs időtúllépési ideje (timeout). Ha a pufferben felszabadul egy üres hely, úgy az adatelemet a pufferbe helyezi, és a műveletet sikeresen befejezettnek nyilvánítja. A művelet sikertelenül fejeződik be, ha a várakozás közben a pufferen *shutdown()* művelet kerülne végrehajtásra.

3.7. definíció (RETRIEVE MŰVELET). A *retrieve*: $C(uid, T, max, E) \to \mathbb{E} \times C(uid, T, max, E) \times T \cup V_{cs'}$ műveletet *retrieve műveletnek* nevezünk, ahol

- végrehajtunk egy, a termelő-fogyasztó problémában ismert *remove()* lépéssorozatot. Ha ez a művelet sikeresen befejeződik, úgy $(x, E') \hookleftarrow cs.E$, $cs' \coloneqq C(cs.uid, cs.T, cs.max, E')$, $e \coloneqq e_{ok}$,
- különben $e \coloneqq e_{err}$, $x \coloneqq undefined$.

3.8. megjegyzés. A termelő-fogyasztó *remove()* művelete során a fogyasztó elsősorban megszerzi a kizárólagos hozzáférést a pufferhez. Amennyiben az üres, úgy várakozni

kezd mindaddig, míg a pufferben legalább egy elem be nem kerül. Ezen várakozásnak jelen esetben nincs időtúllépési ideje. Ha a pufferbe egy elem bekerül, úgy az adatelemet eltávolítja a pufferből, és a műveletet sikeresen befejezettnek nyilvánítja. A művelet azonnal befejeződik sikertelenül, ha a várakozás közben a pufferen *shutdown()* művelet kerülne végrehajtásra.

3.3. Az input protokoll specifikációja

Az input protokollok valamely csatornákon keresztül érkező adatsorozatokat kezelik, és képezik le az adott funkcionális nyelvi elemekre. A csatornákon mozgó jelsorozatok nem csak adatelemeket, hanem csatornavezérlő szimbólumokat is tartalmaznak, és ezen sorozat egy dimenziós listának tekinthető. A protokoll feladata, hogy ezt megfelelő dimenziós listába, listák listájába alakítsa vissza, és a beágyazott kifejezés számára paramétersorozattá alakítsa át.

3.3.1. Előkészület az input protokoll specifikációs leírásához

3.3. jelölés (Egyszerű értéksorozat). Legyen $S_a \coloneqq \langle a_0, a_1, \dots \rangle, a_i \in \tau_r$ átvihető típusú elemek (akár végtelen) sorozata.

3.4. jelölés (protokoll értéksorozat). Legyen $S_a^{(1)} \coloneqq S_a$, továbbá $j \geqq 2$ esetén $S_a^{(j)} \coloneqq \langle a_0, a_1, \dots \rangle$, $a_i \in \bigcup_{k=1}^{j-1} S_a^{(k)}$. Legyen $S_A \coloneqq \langle a_0, a_1, \dots \rangle$, $a_i \in \bigcup_{k=1}^{\infty} S_a^{(k)}$ átvihető típusú elemekből képzett tetszőleges mélységű sorozatok sorozata.

3.9. megjegyzés. Az S_a sorozat elemei csak átvihető típusok lehetnek, az S_A sorozat elemei azonban akár sorozatok sorozata is lehet, tetszőleges mélységben. A protokoll függvények ugyanis képesek listák listáját képezni, ezért erre a jelölésre szükség lesz.

3.5. jelölés (Hibát tartalmazó sorozat). Legyen $S_{A'} \coloneqq \langle a_0, a_1, \dots \rangle, a_i \in S_A \cup \cup \{undefined\}$ halmaz.

3.10. megjegyzés. A csatorna olvasási és értelmezési műveletek hiba esetén nem adják meg a csatornán olvasott adatsorozatot, helyette egy *undefined* értéket generálnak. Ezért az $S_{A'}$ halmazban szerepelhetnek hibátlan elemek (sorozatok), vagy ha valamely olvasási művelet hibázna, az adott sorozatot *undefined* érték helyettesítheti.

3.8. definíció (csatorna olvasó függvény). A $readChannel\colon C(uid, T, max, E) \to \langle T_{cs'} \rangle \times C(uid, T, max, E) \cup \{undefined\}$ függvényt *csatorna olvasó* függvénynek nevezünk, ahol $readChannel(C) = (data, C')$, és

- ha $i = 0$, akkor $(e_i, cs_i, a_i) \coloneqq C.retrieve()$,

- ha $i > 0 \wedge e_{i-1} = e_{ok} \wedge a_{i-1} \neq EoC$, akkor $(e_i, cs_i, a_i) \coloneqq cs_{i-1}.retrieve()$,
- ha $i > 0 \wedge (e_{i-1} \neq e_{ok} \vee a_{i-1} = EoC)$, akkor $(e_i, cs_i, a_i) \coloneqq (e_{err}, cs_{i-1}, undefined,)$,

végül $H \coloneqq \{j \colon j \in \mathbb{N}, a_j = EoC \vee a_j = undefined\}$,

- ha $H = \emptyset$, akkor $data \coloneqq \langle a_0, a_1, \dots \rangle$ és $C' \coloneqq undefined$,
- különben $m \coloneqq min(H), data \coloneqq \langle a_0, a_1, \dots, a_{m-1} \rangle, C' \coloneqq cs_{m-1}$.

3.11. megjegyzés. A képzett sorozat megfelel annak a sorozatnak, amelyet a csatornán adott sorrendben végrehajtott *retrieve()* művelet ad. Amennyiben valamely olvasási művelet nem sikerül, az adott sorozatelem *undefined* lesz, és minden rákövetkező elem is. Amennyiben az *EoC* csatornavég jel is beolvasásra kerül, ezen jel is bekerül a sorozatba, a rákövetkező elemek mindegyike *undefined* elem lesz. Tehát a tényleges olvasás befejeződik az *EoC* szimbólum olvasásakor, vagy az első hiba után. Ha a képzett sorozat véges, akkor a hibamentes olvasást azt jelzi, hogy a sorozat utolsó eleme *EoC* szimbólum-e. Végtelen sorozat mindenképpen hibamentes olvasást jelöl. A függvény visszatérési értéke ezen felül az a csatornaállapot, amelyet az utolsó tényleges olvasás után tartalmaz a rendszer. Végtelen sorozat esetén ez a csatornaállapot nem meghatározható.

A továbbiakban szükségünk lesz a következő függvényekre:

3.9. definíció (SOROZAT ELLENŐRZŐ). Legyen $f_e \colon \langle T_{cs'} \rangle \to \mathbb{B}$, ahol $f_e(data) = b$, $data = \langle a_0, a_1, \dots \rangle$, és

- $b \coloneqq false$, ha $undefined \in data$,
- $b \coloneqq true$, minden más esetben.

3.12. megjegyzés. A függvény sorozatellenőrző szerepkört tölt be. Egy képzett sorozat hibás, ha szerepel benne az *undefined* érték. Minden más esetben hibátlan. Ez utóbbi esetben a *data* lehet végtelen és véges sorozat is.

3.10. definíció (INDEXKIGYŰJTŐ). Legyen az $indexL \colon \langle V_{cs} \rangle \times T_{cs'} \to \langle \mathbb{N} \rangle$, ahol $indexL(data, sign) = ind$, $data = \langle d_0, d_1, \dots \rangle$, és $ind \coloneqq \langle p \colon p \in \mathbb{N} \wedge d_p = sign \rangle$, és teljesül, hogy ha $ind = \langle p_0, p_1, \dots \rangle$, akkor $\forall i, j \in [0, sizeof(ind) - 1], i < j$ esetén $p_i < p_j$.

3.13. megjegyzés. A *indexL* kigyűjti az adatsorozatból azokat az indexeket, amelyek az adott (*sign*) elemekkel egyenlőek. A kigyűjtött indexek szigorúan monoton növekvő sorozatot alkotnak. (A *sign* vagy az *EoS*, vagy az *EoL*, vagy az *EoC* szimbólum lehet.)

3.14. megjegyzés. Egyetlen adatelemet (pl. `Int`) szállító csatorna pufferébe kerülő szimbólumok helyes sorozata kételemű:

$$\langle d, EoC \rangle.$$

(d az adatot jelöli, az adatelem után rögtön *EoC* lezáró végjel következik.)

3.15. megjegyzés. Egyetlen listát (pl. `[Int]`) szállító csatornán a szimbólumok helyes sorozata az alábbihoz hasonló:

$$\langle d, d, d, \dots d, EoS, EoC \rangle.$$

(Adatelemek véges sorozata, majd EoS allista végjel, és rögtön EoC lezáró végjel.)

3.16. megjegyzés. Listák listáját (pl. `[[Int]]`) szállító csatornán a szimbólumok helyes sorozata az alábbihoz hasonló:

$$\langle d, d, \dots d, EoS, EoS, d, d, \dots d, EoS, d, d, \dots d, EoS, EoL, EoC \rangle.$$

(Adatelemek sorozatait EoS allista végjel zárja, a végén az EoL listavég szimbólum, majd az EoC lezáró végjel található.)

3.11. definíció (Részlista előállítás). Legyen $subList\colon \mathbb{N} \times \mathbb{N} \times \langle T_{cs'} \rangle \to \langle T_{cs'} \rangle$, $subList(l, h, data) = D$ függvény, ahol $data = \langle d_0, d_1, \dots \rangle$, $0 \leqq l$, $h < sizeof(data)$ és $l \leqq h$. Ekkor $D \coloneqq \langle a_0, a_1, \dots, a_{n-1} \rangle$ ahol $n = h - l, \forall i \in [0, n-1]$ esetén $a_i \coloneqq d_{h+i}$.

3.17. megjegyzés. A $subList$ függvény az *alsó* (l=low) és *felső* (h=high) indexű elemek közötti részlistát állítja elő. Amennyiben $l = h$, úgy az eredmény egy üres lista lesz. Amennyiben $h = \infty$, úgy az előállított részlista az l. indextől indulva az eredeti lista összes maradék elemét tartalmazza.

3.12. definíció (Index alapú darabolás). A $splitL\colon \langle \mathbb{N} \rangle \times \langle T_{cs'} \rangle \to \langle \langle T_{cs'} \rangle \rangle$, $splitL(ind, data) = D$ legyen olyan függvény, ahol ha $ind = \langle i_0, i_1, \dots \rangle$, $data = \langle d_0, d_1, \dots \rangle$, $n \coloneqq sizeof(ind)$ (n lehet véges vagy végtelen), akkor $D \coloneqq \langle D_0, D_1, \dots \rangle$ az alábbiak szerint: $\forall j \in [0, n-1]$ esetén

- ha $j = 0$, akkor $D_j \coloneqq subList(0, i_0 - 1, data)$,
- ha $j > 0$, akkor $D_j \coloneqq subList(i_{j-1} + 1, i_j - 1, data)$.

3.18. megjegyzés. A $splitL$ egy sorozatot részsorozatokra vág a megadott indexek mentén. Az indexek jellemzően valamely csatorna-vezérlő szimbólum előfordulásainak indexei (pl. EoL). A részsorozatokba a vágási pontokon lévő elem (EoL) nem kerül bele. Az eredmény a részsorozatokból képzett sorozat lesz. A vágás során valamely vágási index lehet ∞ értékű is, ekkor ezen képzett részsorozat az adott kiindulási ponttól kezdve az összes maradék elemet tartalmazni fogja.

3.6. jelölés (Eredmény halmaz). Jelölje a T_{er} a következő halmazt: $T_{er} \coloneqq T_{cs'} \cup \cup \langle T_{cs'} \rangle \cup \langle \langle T_{cs'} \rangle \rangle$.

3.19. megjegyzés. A T_{er} halmaz egy deszerializáció lehetséges eredményhalmaza. A szimbólumsorozat vagy egyetlen elemet, vagy egy listát, vagy listák listáját definiál.

3.13. definíció (Sorozat deszerializáció). Legyen a $collect\colon \langle T_{cs'}\rangle \to T_{er}$, $collect(data) = D$ olyan függvény, ahol ha $data = \langle d_0, d_1, \dots\rangle$ szimbólumok sorozata, és kigyűjtjük a vezérlő szimbólumok indexeit az alábbiak szerint: $ind_{EoL} \coloneqq indexL(EoL, data) = \langle l_0, l_1, \dots\rangle$ az EoL szimbólumok indexei, $ind_{EoS} \coloneqq indexL(EoS, data) = \langle s_0, s_1, \dots\rangle$ az EoS szimbólumok indexei, $ind_{EoC} \coloneqq indexL(EoC, data) = \langle c_0, c_1, \dots\rangle$ az EoC szimbólumok indexei, akkor

- ha $f_e(data) = true$, akkor $D \coloneqq undefined$,
- ha $sizeof(ind_{EoL}) = 1$ és $sizeof(ind_{EoC}) = 1$ és $l_0 = c_0 - 1$ és $l_0 - 1 \in ind_{EoS}$, akkor $D \coloneqq splitL(ind_{EoS}, data)$,
- ha $sizeof(ind_{EoS}) = 1$ és $sizeof(ind_{EoC}) = 1$ és $s_0 = c_0 - 1$, akkor $D \coloneqq subList(0, s_0 - 1, data)$,
- ha $sizeof(ind_{EoC})s = 1$ és $c_0 = 1$ és $sizeof(ind_{EoL}) = 0$ és $sizeof(ind_{EoS}) = 0$, akkor $D \coloneqq d_0$,
- $D \coloneqq undefined$ minden más esetben.

3.20. megjegyzés. Ha hibás a sorozatunk, akkor a *collect* eredménye $undefined$. Ha a sorozatban pontosan egy *EoL - End of Sublist* fordul elő, akkor a sorozat listák listáját írja le, ekkor a *splitL* függvény segítségével az *EoS* szimbólumok mentén kell az adatsorozatot részsorozatokra bontani. Az *EoL* a sorozatban legfeljebb egyszer fordulhat elő. Ha a sorozatban nem szerepel *EoL*, de *EoS - End of Sublist* szerepel (és pontosan egyszer), közvetlenül az *EoC* előtt, akkor a sorozat egyetlen listát ír le. Ekkor az eredményt az *EoS* előtti adatelemek alkotják. Ha sem *EoS*, sem *EoL* nem szerepel a sorozatban, akkor a sorozat egyelemű adatot ír le. Ekkor a második jel az *EoC* kell legyen. Ha így van, akkor a sorozat első eleme adja meg az eredményt. Minden más esetben hibás a sorozat felépítése.

3.7. jelölés (Csatorna tartalma). Jelölje T'_{cs} a $T_{cs'} \cup \langle T_{cs'}\rangle \cup \langle\langle T_{cs'}\rangle\rangle$ halmazt.

3.14. definíció (Csatorna deszerializáció). A $collectChannel\colon C(uid, T, max, E) \to T'_{cs} \times C(uid, T, max, E)$, $collectChannel(C) = (D, C')$ legyen olyan függvény, ahol ha $(S, C') = readChannel(C)$, akkor $D \coloneqq collect(S)$.

3.21. megjegyzés. A *collectChannel* függvény a *collect* függvény azon változata, amely paramétereként nem a csatornáról olvasott szimbólumok sorozatát kapja meg, hanem a csatornát magát. A visszatérési értéke a csatornáról olvasott adatsorozat:

egy elem, egy sorozat, vagy sorozatok sorozata. Ha az olvasás hibás, akkor egyetlen *undefined* érték. A függvény visszatérési értéke ezen felül az olvasás befejezésekor fennálló csatornaállapot.

3.3.2. Az input protokoll definíciója

Az input protokoll n darab csatornát kezel, kiolvassa az általuk hordozott szimbólumok sorozatát, és azokat értelmezi. A csatornavezérlő szimbólumok segítségével építi vissza a saját oldalán az eredeti adatstruktúrát (ez megfelel a programozásban ismert *deszerializáció* fogalmának). Az input protokoll ezen felül működésének megfelelően a helyreállított adatokat még tovább manipulálhatja, pl. a *joink* protokoll listába fűzi őket.

3.15. definíció (INPUT PROTOKOLL ALAKJA). Az $f_{Ihandle}\colon \langle C(uid,T,max,E)\rangle \to S_{A'} \times \langle C(uid,T,max,E)\rangle$, $f_{Ihandle}(cs) = (data, cs')$ alakú függvényeket *input protokoll kezelő* függvényeknek nevezzük, ahol $cs = \langle c_0, c_1, \ldots, c_{n-1}\rangle$ a csatornák egy véges sorozata, $data = \langle a_0, a_1, \ldots, a_{k-1}\rangle$ véges sorozat, melynek elemei (akár végtelen) sorozatok, $cs' = \langle c'_0, c'_1, \ldots, c'_{n-1}\rangle$ a feldolgozás utáni csatornaállapotok.

3.22. megjegyzés. Az input protokoll feladata a konkrét bejövő csatornák adatsorozata alapján bejövő adatokat képezni. Ha a csatornák száma n, a protokollnak lehetősége van azt eltérő, k számú listára leképezni. Az egyes adatelemek lehetnek sorozatok, sorozatok sorozata, stb. A beolvasás megváltoztatja a csatornák állapotait, így az új csatornák is a függvény visszatérési értékei.

3.16. definíció (MEMORY PROTOKOLL). Az

$$f_{memprotI}\colon \langle C(uid,T,max,E)\rangle \to S_{A'} \times \langle C(uid,T,max,E)\rangle,$$

$f_{memprotI}(C) = (D, C')$ függvényt memory *input protokollkezelő függvénynek* nevezzük, ahol ha $C = \emptyset$ (a bemenő csatornák halmaza üres kell legyen), akkor $D \coloneqq \emptyset$ (a képzett értékek sorozata üres sorozat), $C' \coloneqq C$.

3.17. definíció (JOIN1 PROTOKOLL). A $f_{join1prot}\colon \langle C(uid,T,max,E)\rangle \to S_{A'} \times \times\langle C(uid,T,max,E)\rangle$, $f_{join1prot}(C) = (D, C')$ függvényt *join1 input protokollkezelő függvénynek* nevezzük, ahol ha $C = \langle c_0, c_1, \ldots, c_{n-1}\rangle$, akkor $D \coloneqq \langle d_0, d_1, \ldots, d_{n-1}\rangle$, $C' \coloneqq \langle c'_0, c'_1, \ldots, c'_{n-1}\rangle$, ahol $\forall i \in [0, n-1]$ esetén $(d_i, c'_i) \coloneqq collectChannel(c_i)$.

3.23. megjegyzés. Vagyis az eredmény adatsorozat i. elemének típusa, és tartalma megegyezik az i. csatorna típusával, és a róla beolvasható adattokkal. A csatorna olvasás és feldolgozás közbeni hiba esetén az adatsorozat értéke *undefined* lesz.

3.18. definíció (JOINK PROTOKOLL). A $f_{joinkprot}$: $\langle C(uid, T, max, E)\rangle \rightarrow S_{A'} \times \times\langle C(uid, T, max, E)\rangle$, $f_{joinkprot}(C) = (D, C')$ függvényt *joink input protokollkezelő függvénynek* nevezzük, ahol ha $C = \langle c_0, c_1, \ldots, c_{n-1}\rangle$, és teljesül, hogy $\forall i, j \in [0, n - - 1]$ esetén $c_i.T = c_j.T$, akkor $D \coloneqq \langle\langle d_0, d_1, \ldots, d_{n-1}\rangle\rangle$, ahol $\forall i \in [0, n-1]$ esetén $(d_i, c'_i) \coloneqq collectChannel(c_i)$.

3.24. megjegyzés. A *joink* protokoll működéséhez szükséges, hogy minden csatorna típusa azonos legyen. A csatornákon érkező sorozatokat egyetlen sorozatok sorozatává fűzzük össze.

3.3.3. Az input protokoll *wrapper* bemutatása

Az input protokoll által előállított adatok nem feltétlenül illeszkedik a kifejezés paraméterezésére. A kifejezés igényelhet még *helyreállítható* adatokat is, melyek nem a csatornákon érkeztek. Ezen adatokat a D-Box működésétől független (az adott funkcionális nyelvtől függő módon) kell előállítani. A helyreállítható adatok sorozatát, és az input protokoll által beolvasott értékeket az *input wrapper* fésüli össze, és illeszti rá a kifejezés paraméterezésére.

3.19. definíció (INPUT WRAPPER). Legyen $wrapInp$: $\langle T_\omega\rangle \times S_{A'} \times \langle\delta\rangle \rightarrow \langle\tau' \cup \delta\rangle$, $wrapInp(ed, et, w) = S$ függvényt. Ezt *input wrapper* függvénynek nevezzük, ahol ha $et = \langle t_0, t_1, \ldots, t_{k-1}\rangle$ típusok sorozata, $ed = \langle d_0, d_1, \ldots, d_{n-1}\rangle$ csatornaadatok sorozata, $w = \langle w_0, w_1, \ldots, w_{m-1}\rangle$ helyreállítható értékek sorozata, és teljesül, hogy $k = n + m$, és $\nexists d \in ed, d = undefined$, akkor $S \coloneqq \langle s_0, s_1, s_2, \ldots, s_{k-1}\rangle$ az alábbi módon: legyen $ed_0 \coloneqq ed$, $et_0 \coloneqq et$, és $\forall i \in [0, k-1]$ esetén

- ha $t_i \in \tau'$, akkor $(s_i, ed_{i+1}) \hookleftarrow ed_i$, $et_{i+1} \coloneqq et_i$,
- ha $t_i \in \delta$, akkor $(s_i, et_{i+1}) \hookleftarrow et_i$, $ed_{i+1} \coloneqq ed_i$.

3.25. megjegyzés. Az *wrapInp* megkapja a kifejezés paramétereinek típussorozatát (et), az input csatornákon beérkező adatok a protokoll által feldolgozott sorozatát (ed), valamint helyreállítható értékek sorozatát (w). Eredményül adja az ed és w sorozatok összefésülésének eredményét, mely összefésülést a típusok jellege vezérli. Ha az argumentum típusa valamely kezelhető típus, akkor az érték a csatornákról érkező értékek következő eleme lesz, ellenkező esetben a helyreállítható értékek következő eleme kerül be. Ennek alapján a wrapper függvény képes előállítani a kifejezés által igényelt k darab argumentum értéket.

A *wrapper* nincs értelmezve, ha az olvasott adatsorozat valamelyike hibás volt, $undefined$ értékű.

3.20. definíció (INPUT PROTOKOLL). Az f_{Iprot}: $f_{Ihandle} \times \langle C(uid, T, max, E)\rangle \times \times\langle T_\omega\rangle \times \langle\delta\rangle \rightarrow \langle\omega\rangle \times C(uid, T, max, E)$, $f_{Iprot}(f, C, T, w) = (D, C'), f \in \{f_{memprotI},$

$f_{join1prot}, f_{joinkprot}\}$ függvényt *input protokollnak* nevezzük, ahol $C = \langle c_0, c_1, \ldots, c_{n-1}\rangle$, $T = \langle t_0, t_1, \ldots, t_{k-1}\rangle$ esetén $(data, C') \coloneqq f(C)$, $D \coloneqq wrapInp(T, data, w)$.

3.26. megjegyzés. Az *input protokoll* megkapja a konkrét input protokollkezelő függvényt, a doboz input csatornáinak listáját, a kifejezés paramétereinek típuslistáját és a helyreállítható típusú értékek sorozatát. Alkalmazza a csatornákra az input protokollkezelő függvényt, majd az *input wrapper* függvény segítségével az eredménylistába beilleszti a helyreállítható típusú értékeket.

3.4. Az output protokoll specifikációja

Az output protokoll feladata, hogy a kifejezés által előállított, akár több dimenziós adatokból felépített sorozatát egy dimenziós, a megfelelő pozíciókon csatornavezérlő szimbólumokat is tartalmazó sorozattá alakítsa át (ez a *szerializáció* művelete). Ezen felül a szerializálás eredményeképpen kapott szimbólumsorozatot a csatornákra továbbítsa. Figyelni kell arra, hogy a kifejezés eredménye helyreállítható típusú értékeket is tartalmazhat, melyeket nem lehet a csatornákra kiírni, azokat el kell távolítani a feldolgozás előtt.

3.21. definíció (OUTPUT PROTOKOLL ALAKJA). Az

$$f_{Ohandle}\colon \langle\langle T_{cs'}\rangle\rangle \times \langle C(uid, T, max, E)\rangle \to \langle C(uid, T, max, E) \cup \{undefined\}\rangle,$$

$f_{Ohandle}(S, C) = C'$ alakú függvényeket *output protokoll kezelő* függvényeknek nevezzük, ahol $S = \langle s_0, s_1, \ldots, s_{n-1}\rangle$ csatornára írható szimbólumok sorozatai, $C = \langle c_0, c_1, \ldots, c_{n-1}\rangle$ csatornák egy sorozata, C' pedig a kiírás utáni csatornaállapotok sorozata.

3.4.1. Az output protokoll szemantikai leírásának előkészítése

Az alábbiakban olyan segédfüggvényeket definiálunk, melyekre a konkrét szemantikai leírás során hivatkozni fogunk.

3.22. definíció (E SZERIALIZÁCIÓ). Legyen $symbolsE\colon \tau_r \to S_A$, $symbolsE(d) = r$, és $r \coloneqq \langle d\rangle$.

3.27. megjegyzés. A $symbolsE$ egyetlen adathoz generálja a csatornára írandó szimbólumsorozatot, mely csak magát az adatelemet tartalmazza.

3.23. definíció (L SZERIALIZÁCIÓ). Legyen a $symbolsL\colon \langle\tau_r\rangle \to \langle T_{cs'}\rangle$ olyan függvény, ahol $symbolsL(data) = r$ esetén ha $data = \langle d_0, d_1, \ldots\rangle$, akkor $r \coloneqq data \oplus EoS$.

3.28. megjegyzés. A $symbolsL$ függvény generálja a csatornára írandó szimbólumsorozatot egy dimenziós lista esetén. A lista elemek kiírása után egy EoS szimbólummal jelezzük a lista végét. Amennyiben az adatlista végtelen számosságú, ez a jel a gyakorlatban sosem íródik majd ki a csatornára, de ekkor a feldolgozó doboz amúgy is vélhetően az adatelemek véges sorozatát fogja csak feldolgozni. Amennyiben a lista üres (nulla elemszámú), úgy az eredmény sorozat egyetlen EoS szimbólumból áll.

3.24. definíció (LL szerializáció). Legyen a $symbolsLL\colon \langle\langle\tau_r\rangle\rangle \to \langle T_{cs'}\rangle$ olyan függvény, ahol $symbolsL(data) = r$ esetén ha $data = \langle d_0, d_1, \ldots, d_{n-1}\rangle$, akkor $r \coloneqq symbolsL(d_0) \oplus symbolsL(d_1) \oplus \cdots \oplus symbolsL(d_{n-1}) \oplus EoL$.

3.29. megjegyzés. A $symbolsLL$ függvény listák listájához rendel szimbólumsorozatot. Az allisták szimbólumsorozatát (melyek EoS-el vannak lezárva) egy EoL szimbólummal egészíti ki. Amennyiben valamely allista végtelen, úgy ez a kiegészítés a gyakorlatban csak elvi, mivel csatornára írás során az allisták feldolgozása adott sorrendben történik, a következő allista szimbólumsorozatának kiírása csak akkor kezdődik majd el, ha az előző minden szimbóluma már kiíródott. Vagyis ha egy d_i sorozat végtelen, a i. sorozat után egyetlen sorozat egyetlen szimbóluma sem fog kiíródni, sem az EoL lezáró elem.

3.25. definíció (szerializáció). Legyen a $symbolsAll\colon \tau \to \langle T_{cs'}\rangle$ olyan függvény, ahol $symbolsAll(data) = ser$ esetén ha $data = \langle d_0, d_1, \ldots, d_{n-1}\rangle$, akkor

- ha $depth(data) = 0$, akkor $ser \coloneqq symbolsE(data) \oplus EoC$,
- ha $depth(data) = 1$, akkor $ser \coloneqq symbolsL(data) \oplus EoC$,
- ha $depth(data) = 2$, akkor $ser \coloneqq symbolsLL(data) \oplus EoC$,
- egyénként $ser \coloneqq \langle EoC\rangle$.

3.30. megjegyzés. A $symbolsAll$ függvény elem, lista, listák listája esetén megadja a csatornára írandó szimbólumok sorozatát. Az adatok sorozatát egy dimenziós sorozattá alakítja, megfelelő pozíciókon beszúrva a csatorna vezérlő szimbólumokat, a végére illesztve a lezáró EoC szimbólumot. Az előállt sorozat elemeit adott sorrendben kell a csatornára kiírni, de ez már a konkrét protokoll függvény feladata.

3.26. definíció (csatornára írás). A $writeToChannel\colon \langle T_{cs'}\rangle \times C(uid, T, max, E) \to C(uid, T, max, E) \cup \{undefined\}$, $writeToChannel(S, C) = C'$ legyen olyan függvény, ahol ha $S = \langle s_0, s_1, \ldots\rangle$, és $\forall i \in \mathbb{N}$ esetén

- ha $i = 0$, akkor $(e_0, c_0) \coloneqq C.store(s_0)$,

- ha $i > 0 \wedge e_{i-1} = e_{ok} \wedge s_{i-1} = EoC$, akkor $(c_i, e_i) \coloneqq (c_{i-1}, e_{finished})$,
- ha $i > 0 \wedge e_{i-1} = e_{finished}$, akkor $(c_i, e_i) \coloneqq (c_{i-1}, e_{finished})$,
- ha $i > 0 \wedge e_{i-1} = e_{ok} \wedge s_{i-1} \neq EoC$, akkor $(c_i, e_i) \coloneqq c_{i-1}.store(s_i)$,
- ha $i > 0 \wedge e_{i-1} \neq e_{ok} \wedge e_{i-1} \neq e_{finished}$, akkor $(c_i, e_i) \coloneqq (c_{i-1}, e_{i-1})$,

akkor legyen $H \coloneqq \{i \colon i \in \mathbb{N}, e_i = e_{finished}\}$, ha $sizeof(H) \neq 0$ akkor $m \coloneqq min(H)$, $C' \coloneqq c_m$, ellenkező esetben $C' \coloneqq undefined$.

3.31. megjegyzés. A *writeToChannel* függvény a csatornára írandó szimbólumok sorozatát írja ki. A sikeres *EoC* kiírás után további írási műveletek nem történnek. Amennyiben a kiírandó szimbólumok sorozata végtelen, sikeres *EoC* írás nem következhet be. A függvény visszatérési értéke a sikeres *EoC* írási műveletkori csatorna állapot. Amennyiben ilyen nincsen, gyakorlati visszatérési érték feldolgozására nem kerül sor, ezért *undefined* érték lesz a visszatérési érték.

A *memory* output protokoll szemantikája egyszerű: nem dolgoz fel adatokat, és nem továbbít egyetlen csatornára sem szimbólumokat.

3.27. definíció (MEMORY PROTOKOLL). Legyen az $f_{memprotO} \colon \langle\langle T_{cs'}\rangle\rangle \times \langle C(uid, T, max, E)\rangle \to \langle C(uid, T, max, E)\rangle$, $f_{memprotO}(S, C) = C'$ függvényt *memory output protokollkezelő* függvénynek nevezzük, ahol ha $S = \emptyset$, $C = \emptyset$, akkor $C' \coloneqq C$.

3.32. megjegyzés. A *memory* output protokoll esetén nincsenek csatornák, sem adatok. Mivel nincsenek csatornák, azok állapotai sem képesek megváltozni.

A *split1* protokoll szemantikája szerint n darab adatsorozatot és n darab csatornát kezel, minden adatsorozatot a megfelelő csatornára továbbítja.

3.28. definíció (SPLIT1 PROTOKOLL). Legyen az $f_{split1prot} \colon \langle\langle T_{cs'}\rangle\rangle \times \langle C(uid, T, max, E)\rangle \to \langle C(uid, T, max, E) \cup \{undefined\}\rangle$, $f_{split1prot}(S, C) = C'$ függvény. Ezt *split1 output protokollkezelő*nek nevezzük, ahol ha $S = \langle s_0, s_1, \ldots, s_{n-1}\rangle$ szimbólumsorozatok sorozata, $C = \langle c_0, c_1, \ldots, c_{n-1}\rangle$ csatornák, akkor $C' \coloneqq \langle c'_0, c'_1, \ldots, c'_{n-1}\rangle$, és $\forall i \in [0, n-1]$ esetén $c'_i \coloneqq writeToChannel(s_i, c_i)$.

A *splitk* protokoll egyetlen szimbólumsorozatot, és n darab csatornát kezel. A szimbólumsorozatnak n darab allistából kell állnia, különben a *splitk* protokoll nem képes azt feldolgozni. Minden egyes allistát egyetlen csatornához rendeli, majd a szimbólumsorozatot a választott csatornára továbbítja.

3.29. definíció (SPLITK PROTOKOLL). Legyen $f_{splitkprot} \colon \langle\langle T_{cs'}\rangle\rangle \times \langle C(uid, T, max, E)\rangle \to \langle C(uid, T, max, E) \cup \{undefined\}\rangle$, $f_{splitkprot}(S, C) = C'$. Ezt *splitk output protokollkezelő* függvénynek nevezzük, ahol ha $S = \langle s_0\rangle$ egy elemű adatsorozat, $C = \langle c_0, c_1, \ldots, c_{n-1}\rangle$ csatornák sorozata, és teljesül, hogy $s_0 = \langle s'_0, s'_1, \ldots, s'_{n-1}\rangle$ is n elemű, akkor $C' \coloneqq \langle c'_0, c'_1, \ldots, c'_{n-1}\rangle$, és $\forall i \in [0, n-1]$ esetén $c'_i \coloneqq writeToChannel(s'_i, c_i)$.

3.4.2. A *splitf* output protokoll szemantikája

A *splitf* protokoll működése a legbonyolultabb. A *split1* és *splitk* protokollok esetén egyszerű az a döntés, hogy melyik adatsorozatot melyik csatornára kell továbbítani, mivel végső soron az adatsorozatok száma és a csatornák száma egyezett. A döntést a protokollok indulásakor meg lehetett hozni. A *splitf* protokoll esetén az adatsorozatok száma és a csatornák száma általában nem egyezik meg. A működés során hozza meg a protokoll azt a döntést, hogy melyik adatsorozat melyik csatornára kerüljön. Egy ilyen döntést utólag már természetesen nem módosíthat, hiszen nem tekintjük helyes működésnek, ha egy adatsorozat egyik része egyik, másik része másik csatornára kerül kiírásra.

A döntéshozás módszere szerint az adatsorozatokat és a csatornákat össze kell párosítani:

- kialakítunk egy *figyelt csatornák* listáját, melybe induláskor minden csatornát beleértünk,
- amennyiben van olyan *figyelt csatorna*, amelyhez még nem rendeltünk adatsorozatot (*szabad csatorna*), és van olyan adatsorozat, amelyet még nem rendeltünk csatornához (*szabad sorozat*), úgy e kettőt össze kell rendelni,
- ha van szabad csatorna, de nincs szabad adatsorozat, akkor erre a csatornára a későbbiekben sem fogunk tudni adatsorozatot rendelni, így ezen csatornát eltávolíthatjuk a figyelt csatornák listájáról,
- ha van olyan figyelt csatorna, amely jelenleg képes fogadni adatelemet – mert a puffere nem telített –, akkor a csatornához rendelt adatsorozat következő elemét kiküldjük erre a csatornára,
- ha valamely figyelt csatorna képes adatelemet fogadni, de a hozzá rendelt adatsorozat minden elemét már kiküldtük rá, akkor ezen csatorna a továbbiakban *szabad* csatornának minősül, úgy tekintjük, hogy nincs hozzárendelt adatsorozat.

Egy csatorna puffere nem „telített”, ha a puffer tárolási kapacitásának 90%-os csak a kihasználtsága [1].

A protokoll működésének kezdetekor van n darab *szabad* csatorna, és k darab *szabad* adatsorozat, és (általában) $n < k$ teljesül. A k szabad sorozatot első n elemét a működés elején gyorsan hozzárendeljük a szabad csatornákhoz, és amíg ezen sorozatok kiküldése nem fejeződik be, a további adatsorozatok várnak. Amelyik csatorna először szabadul fel újra, ahhoz kötjük a $k+1$. szabad sorozatot, majd a következő felszabadult csatornához a $k+2$. sorozatot, és így tovább. Ha az adatsorozatok nem végtelenek, akkor idővel minden adatsorozatot csatornához fogunk tudni rendelni.

Vegyük észre, hogy a fenti működés $n > k$ és $n = k$ esetén is végrehajtható. Első

[1] Ha a csatorna legalább 100 elemű, ez esetben legalább 10 elemnyi hely szabad a csatornán – ez a gyakorlatban megfelelőnek bizonyul.

esetben több a csatornánk, mint a sorozataink száma, a párosítások akár statikusan is végrehajthatók lennének. Lesznek olyan csatornák, amelyek semmilyen adatsorozatot nem kapnak meg. Második esetben a csatornák száma és a sorozatok száma pontosan megegyezik, mely esetben minden csatornához egy adatsorozatot köthetünk. Ez lényegében megegyezik a *splitk* protokoll működésével.

3.30. definíció (PUFFER NEM TELÍTETT). Legyen az $isLow\colon C(uid, T, max, E) \to \mathbb{B}$, $isLow(cs) = r$ függvény, ahol ha $sizeof(cs.E) < 0.9 * cs.max$, akkor $r \coloneqq true$, különben $r \coloneqq false$. Az $isLow(cs)$ függvényt $cs.isLow()$ alakban fogjuk használni.

3.33. megjegyzés. Az $isLow()$ függvény meghatározza, hogy a csatorna pufferének aktuális terhelése 90% alatti-e vagy sem.

3.31. definíció (CSATORNA SZABAD). A $f_{free}\colon \langle C(uid, T, max, E)\rangle \to \langle C(uid, T, max, E)\rangle$, $f_{free}(C) = C'$ legyen olyan függvény, ahol $C' \coloneqq \langle c\colon c \in C \wedge c.isLow()\rangle$.

3.34. megjegyzés. Az f_{free} függvény csatornák egy sorozata alapján megadja, melyek azok a csatornák, melyek puffere nincs teljesen tele (van benne szabad hely).

3.32. definíció (SZABADRA VÁR). Legyen $f_{waitFree}\colon \langle C(uid, T, max, E)\rangle \to \langle C(uid, T, max, E)\rangle$, $f_{waitFree}(C) = C'$, $C = \langle c_0, c_1, \ldots, c_{n-1}\rangle$, és $C_s \coloneqq \langle C'_0, C'_1, \ldots\rangle$, ahol $\forall i \in \in [0, \ldots]$ esetén

- ha $i = 0$, akkor $C'_0 \coloneqq f_{free}(C)$,
- ha $i > 0 \wedge sizeof(C'_{i-1}) = 0$, akkor $C'_i \coloneqq f_{free}(C'_{i-1})$,
- ha $i > 0 \wedge sizeof(C'_{i-1}) > 0$, akkor $C'_i \coloneqq C'_{i-1}$,

valamint $m \coloneqq min\{j\colon j \in N, C'_j = C'_{j+1} \wedge sizeof(C'_j) > 0\}$, ekkor $C' \coloneqq C'_m$.

3.35. megjegyzés. A $f_{waitFree}$ függvény *megvárja*, amíg lesz legalább egy olyan csatorna, amely képes adatokat fogadni. A visszatérési értéke nem üres sorozat, legalább egy csatornát tartalmaz.

3.33. definíció (RANDOM). Legyen $f_{random}\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ olyan véletlen számot generáló függvény, melyre teljesül, hogy $f_{random}(L, H)$ diszkrét egyenletes eloszlású az $\{L, L+1, \ldots, H\}$ halmazon, minden $L, H \in \mathbb{N}$, $L < H$ esetén.

3.36. megjegyzés. Az f_{random} függvény az $[L, H]$ intervallumból véletlenszerű egész számot generál.

3.34. definíció (SZABADOT VÁLASZT). Az $f_{chooseFree}\colon \langle C(uid, T, max, E)\rangle \to C(uid, T, max, E)$, $f_{chooseFree}(C) = C_f$ legyen olyan függvény, ahol ha $C = \langle c_0, c_1, \ldots, c_{n-1}\rangle$, $sizeof(C) > 0$, és $i \coloneqq f_{random}(0, n-1)$, akkor $C_f \coloneqq c_i$.

3.37. megjegyzés. Az $f_{chooseFree}$ a legalább egy csatornát tartalmazó listáról véletlenszerűen választ egy elemet visszatérési értékként.

3.35. definíció (FREE PÁROS). Az $R(s, c)$ formális kettest *free kommunikációs párosnak* nevezzük, ahol

- $s \in \langle T_{cs'} \rangle$ egy szimbólumsorozat,
- $c \in C(uid, T, max, E) \cup \{undefined\}$ vagy egy csatorna, vagy az *undefined* érték.

3.38. megjegyzés. Valamely $r \in R(s, c)$ *free* kommunikációs páros $r.s$ eleme egy csatornára küldhető szimbólumsorozat, $r.c$ eleme vagy egy konkrét csatorna, vagy az *undefined* érték. Első esetben a szimbólumsorozat már hozzá van rendelve egy konkrét csatornához, a második esetben még nem dőlt el, melyik csatornára fog a sorozat kerülni (csatornaválasztás nem következett még be).

3.36. definíció (SZABAD SOROZAT). Legyen az $f_{setChannel} \colon \langle R(s, c) \rangle \times C(uid, T, max, E) \to \langle R(s, c) \rangle$, ahol $f_{setChannel}(r, c) = r'$ olyan függvény, ahol ha $r = \langle r_0, r_1, \ldots, r_{n-1} \rangle$, $H \coloneqq \{j : j \in [0, n-1], r_j.C = undefined\}$, és

- ha $sizeof(H) = 0$, akkor $r' \coloneqq \langle r_0, r_1, \ldots, r_{n-1} \rangle$,
- ha $sizeof(H) > 0$, akkor $i \coloneqq min(H), r' \coloneqq \langle r_0, r_1, \ldots, r_{i-1}, r'_i, r_{n+1}, \ldots, r_{n-1} \rangle$, és $r'_i.s \coloneqq r_i.s$, $r'_i.C \coloneqq c$.

3.39. megjegyzés. Az $f_{setChannel}$ függvény adott *free* kommunikációs páros sorozatban megkeresi az első olyan párost, amelynél a csatornarész még nincs kitöltve (*undefined*), és beteszi oda a kívánt csatornát. Ezzel a kiválasztott páros szimbólumsorozatát ehhez a csatornához köti. Ha a kommunikációs páros sorozatban már minden szimbólumsorozat csatornához van rendelve, akkor az eredeti kommunikációs páros sorozatot érintetlenül hagyja.

3.37. definíció (INICIALIZÁLÁS). Legyen az $f_{initFree} \colon \langle \langle T_{cs'} \rangle \rangle \to \langle R(s, c) \rangle$ függvény, ahol $f_{initFree}(S_s, C_s) = R_s$ esetén ha $S_s = \langle s_0, s_1, \ldots, s_{n-1} \rangle$ szimbólumsorozatok, akkor $R_s \coloneqq \langle r_0, r_1, \ldots, r_{n-1} \rangle$, és $\forall i \in [0, n-1]$ esetén $r_i \coloneqq (s_i \ominus EoC, undefined)$.

3.40. megjegyzés. A *free* protokoll több szimbólumsorozatot küldhet ki ugyanazon csatornára szükség esetén. A szimbólumsorozatokat jellemzően *EoC* zárja. A kiinduló állapotba lépéskor ezért a szimbólumsorozatokból eltávolítjuk az *EoC* jeleket. Másrészről, induláskor még egyik szimbólumsorozat sincs konkrét csatornához rendelve, ezt jelzi a sorozathoz tartozó *undefined* érték. A függvény eredménye tehát a módosított szimbólumsorozatok, párosítva az *undefined* értékekkel.

3.38. definíció (CSATORNÁT KERES). Az $f_{findR}\colon \langle R(s,c)\rangle \times C(uid,T,max,E) \to R(s,c) \cup \{undefined\}$, $f_{findR}(R_s,C) = E$ függvény, ahol ha $R_s = \langle r_0, r_1, \ldots, r_{n-1}\rangle$, és $H \coloneqq \{i\colon i \in [0,n-1] \wedge r_i.C.I = C.I\}$ esetén

- ha $sizeof(H) = 0$, akkor $E \coloneqq undefined$,
- ha $sizeof(H) > 0$, akkor $j \coloneqq min(H)$, $E \coloneqq r_j$.

3.41. megjegyzés. Az f_{findR} a sorozatból megkeresi azt az első olyan kommunikációs párost, amely az adott csatornához tartozik. Ha ilyen páros nincs, akkor *undefined* értékkel tér vissza.

3.39. definíció (SZABADRA KÜLD). Az $f_{sendFree}\colon \langle R(s,c)\rangle \times C(uid,T,max,E) \to \langle R(s,c)\rangle \times C(uid,T,max,E) \times \mathbb{E}$, $f_{sendFree}(R_s,C) = (R'_s,C',e)$ legyen olyan függvény, ahol $r_i = f_{findR}(R_s,C)$ esetén

- ha $r_i = undefined$, akkor $R'_s \coloneqq R_s, C' \coloneqq C, e \coloneqq e_{ok}$,
- ha $r_i \neq undefined$, akkor $(x, r'_i.s) \hookleftarrow r_i.s$, $(err, c') \coloneqq store(x,C)$,
  - ha $err = e_{ok} \wedge sizeof(r'_i) > 0$ akkor $R'_s \coloneqq \langle r_0, r_1, \ldots r_{i-1}, r'_i, r_{i+1}, \ldots, r_{n-1}\rangle$, $C' \coloneqq c', e \coloneqq e_{ok}$,
  - ha $err = e_{ok} \wedge sizeof(r'_i) = 0$ akkor $R'_s \coloneqq \langle r_0, r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_{n-1}\rangle$, $C' \coloneqq c', e \coloneqq e_{ok}$,
  - ha $err \neq e_{ok}$, akkor $R'_s \coloneqq R_s, C' \coloneqq c, e' \coloneqq e_{err}$.

3.42. megjegyzés. Az $f_{sendFree}$ függvény a paraméterként megkapott csatornára megpróbál elemet küldeni. Először kikeresi a sorozatban, hogy az adott csatornához melyik szimbólumsorozat tartozik, majd ezen sorozat következő elemét elküldi a csatornára. Ha a küldés sikeres, akkor a küldött elemet el is távolítja a sorozatból. Ha ez az elem a sorozat utolsó eleme volt (eltávolítás után üres sorozatot kaptunk), akkor ezt az üres sorozatot tartalmazó kommunikációs párost eltávolítja a párosok sorozatából. Ha a küldés során probléma merült fel, vagy a csatornához nincs hozzárendelve küldésre váró szimbólumsorozat, akkor nem változtat meg semmit a kommunikációs sorozaton. A művelet végén hibakóddal jelzi, hogy történt-e hiba a végrehajtás közben.

3.40. definíció (KÖVETKEZŐ LÉPÉS). Legyen az $f_{oneStep}\colon \langle R(s,c)\rangle \times \langle C(uid,T,max,E)\rangle \to \langle R(s,c)\rangle \times \langle C(uid,T,max,E)\rangle \times \langle C(uid,T,max,E)\rangle$, $f_{oneStep}(R_s,C_s) = (R'_s,C'_s)$ olyan függvény, ahol $R_s = \langle r_0, r_1, \ldots, r_{n-1}\rangle$, $C_s = \langle c_0, c_1, \ldots, c_{n-1}\rangle$, és $C_f \coloneqq f_{waitFree}(C_s)$, $c_r \coloneqq f_{chooseFree}(C_f)$, $i \coloneqq min\{j\colon j \in [0,n-1], r_j = c_r\}$, $rr \coloneqq f_{findR}(R_s,c_r)$ esetén

- ha $rr = undefinef$ akkor $R_r \coloneqq f_{setChannel}(R_s, c_r)$, $(R'_s, c'_r, e_r) \coloneqq f_{sendFree}(R_r, c_r)$, $C'_s \coloneqq \langle c_0, c_1, \ldots, c_{i-1}, c'_r, c_{i+1}, \ldots, c_{n-1}\rangle$,
- ha $f_{findR}(R_s, c_r) \neq undefinef$ akkor $C'_s \coloneqq \langle c_0, c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_{n-1}\rangle$, $R'_r \coloneqq R_r$.

3.43. megjegyzés. Az $f_{oneStep}$ függvény elküldendő sorozatok és csatornák alapján választ egy véletlenszerű, szabad csatornát, és küld rá egy jelet. Amennyiben a véletlenszerű választás során olyan csatornát választottunk volna ki, amelyre nincs ütemezett szimbólumsorozat, valamint már nincs szabad (nem ütemezett) szimbólumsorotat sem, amelyet ehhez a csatornához lehetne rendelni, akkor a függvény nem változtat sem a csatornák állapotán, sem a szimbólumsorozatokon. Ezt a csatornát eltávolítjuk a C_s sorozatból, hogy legközelebb ne válasszuk ki mint szabad csatornát. Az eltávolított csatornát a feldolgozott csatornák listájára (C_f) helyezzük át. A küldési tevékenységet megszakíthatjuk, ha már minden csatornát áthelyeztünk a feldolgozott csatornák listájára.

3.41. definíció (KÜLDÉST BEFEJEZ). Legyen az $f_{sendFinish}\colon C(uid, T, max, E) \rightarrow C(uid, T, max, E)$, $f_{sendFinis}(C) = C'$ olyan függvény, ahol

- ha $depth_T(C.T) = 2$ akkor $C' \coloneqq store(store(C, EoL), EoC)$,
- ha $depth_T(C.T) = 1$ akkor $C' \coloneqq store(store(C, EoS), EoC)$,
- ha $depth_T(C.T) = 0$ akkor $C' \coloneqq store(C, EoC)$,

3.44. megjegyzés. Az $f_{sendFinish}$ függvény egy adott csatornát lezár. A csatorna típusának függvényében megfelelő lezáró jeleket küld ki.

3.42. definíció (CSATORNÁT BEFEJEZ). Az $f_{sendEoC}\colon \langle C(uid, T, max, E)\rangle \rightarrow \langle C(uid, T, max, E)\rangle$, $f_{sendEoC}(C_s) = C'_s$, legyen olyan függvény, ahol $C_s \coloneqq \langle c_0, c_1, \ldots, c_{k-1}\rangle$, és $C'_s \coloneqq \langle c'_0, c'_1, \ldots, c'_{k-1}\rangle$, $\forall i \in [0, n-1]$ esetén $c'_i \coloneqq f_{sendFinish}(c_i)$.

3.45. megjegyzés. Az $f_{sendEoC}$ függvény minden csatornát lezárt az $f_{splitfprot}$ működésének végén, az $f_{sendFinish}$ függvény segítségével.

3.43. definíció (SPLITF PROTOKOLL). Legyen az $f_{splitfprot}\colon \langle\langle T_{cs'}\rangle\rangle \times \langle C(uid, T, max, E)\rangle \rightarrow \langle C(uid, T, max, E) \cup \{undefined\}\rangle$, $f_{splitfprot}(S_s, C_s) = C'_s$ függvény. Ezt *splitf output protokollkezelő* függvényeknek nevezzük, ahol $S_s = \langle s_0, s_1, \ldots, s_{n-1}\rangle$, $C_s = = \langle c_0, c_1, \ldots, c_{k-1}\rangle$, és $R_s \coloneqq f_{initFree}(S_s, C_s)$, $R_r \coloneqq \langle r_0, r_1, \ldots, \rangle$, $\forall i \in [0, ..]$ esetén

- ha $i = 0$, akkor $(R_i, C_i, F_i) \coloneqq f_{oneStep}(R_s, C_s, \emptyset)$,
- ha $i > 0 \wedge sizeof(C_{i-1}) > 0$ akkor $(R_i, C_i, F_i) \coloneqq f_{oneStep}(R_i, C_i, F_i)$,

- ha $i > 0 \wedge sizeof(C_{i-1}) = 0$ akkor $(R_i, C_i, F_i) \coloneqq (R_{i-1}, C_{i-1}, F_{i-1})$,

és legyen $H \coloneqq \{j \colon j \in [0..] \wedge C_j = \emptyset\}$, $m \coloneqq min(H)$, akkor $C'_s \coloneqq f_{sendEoC}(F'_m)$.

3.46. megjegyzés. A $f_{splitfprot}$ függvény a bemeneti szimbólumhalmazt a csatornákra küldi. A függvény minden lépésben kiválaszt egy olyan csatornát, amely nincs tele, fogadáskész, és elküld rá egy jelet. Amennyiben valamely csatornára már nincs küldendő jel, sem hozzárendelhető szimbólumsorozat, úgy a csatornát átrakja a feldolgozott listára. Amennyiben már minden csatorna átkerült a feldolgozott csatornák listájára, úgy a protokoll minden csatornát lezár a típusának megfelelő jelsorozattal.

3.4.3. Az output protokoll wrapper

Az output protokoll wrapper felelős azért, hogy a kifejezés eredményében szereplő *helyreállítható* típusú értékeket a protokoll ne kapja meg, csak a *kezelhető* típusú elemeket.

3.8. jelölés (KIFEJEZÉS OUTPUT). Jelöljük $S_{A*} = \langle a_0, a_1, \dots \rangle, a_i \in S_a \cup S_A \cup \delta$ kezelhető típusú elemekből, vagy helyreállítható típusú elemekből képzett sorozatot.

3.44. definíció (HELYREÁLLÍTHATÓT KIHAGY). Legyen $f_\diamond \colon \langle \tau' \rangle \times \tau' \cup \delta \to \langle \tau' \rangle$, $f_\diamond(S, a) = S'$ függvény, ahol ha $a \in \tau'$, akkor $S' \coloneqq S \oplus a$, különben $S' \coloneqq S$. Az $f_\diamond$ függvényt operátor alakban $S' \coloneqq S \diamond a$ formában fogjuk használni.

3.47. megjegyzés. Az $f_\diamond$ függvény kezelhető típusú értékek sorozatához hozzáfűzi az adott elemet, amennyiben az elem szintén kezelhető típusú. Ha helyreállítható típusú az elem, akkor nem fűzi a sorozat végére.

3.45. definíció (OUTPUT WRAPPER). Legyen $wrapOut \colon \langle \omega \rangle \times S_{A*} \to \langle S_A \rangle$ függvény, $wrapOut(et, ed) = S$. Ezt *output wrapper* függvénynek nevezzük, ahol ha $ed = \langle d_0, d_1, \dots, d_{n-1} \rangle$, $et = \langle t_0, t_1, \dots, t_{k-1} \rangle$, akkor $S \coloneqq d_0 \diamond d_1 \diamond \dots \diamond d_{n-1}$.

3.48. megjegyzés. Az *wrapOut* leképezés paramétereként megkapja a kifejezés outptjának típuslistáját (et), valamint a kifejezés által generált értékek listáját (ed), amelyen akár helyreállítható típusú értékek is szerepelhetnek. Mivel ez utóbbiak nem szállítható értékek, a *wrapper* ezeket az értékeket eltávolítja az ed listáról. Az eredmény listában szereplő értékek mindegyik szállítható típusú, ezért átadhatók az output protokollnak.

3.49. megjegyzés. Az et paraméterre szükség lehet olyan esetben, ha valamely adat típusvizsgálata nem megoldható. Ez esetben az $f_\diamond$ függvény nem csak az adott d_i értéket, hanem annak típusát jelző t_i típust is megkaphatja, hogy segítse a döntést.

3.5. Összefoglalás

Ebben a fejezetben definiáltuk az elem, a lista, a listák listájának szerializációját a csatornavezérlő szimbólumok segítségével. Definiáltuk, hogy milyen csatorna műveletekre van szükség a projekt futása során. Megadtuk, hogy melyik output protokoll milyen módon szerializálja a paraméterül kapott adatsorozatot, hogyan választ számukra csatornát. Megadtuk, hogy az output protokoll wrapper függvény hogyan távolítja el a kifejezés outputjában szereplő helyreállítható típusú értékeket, és adja át a maradékot az output protokollnak. Bemutatásra került, hogy az input protokollok milyen módon értelmezik a csatornákon bejövő jelsorozatot, hogyan deszerializálják azt, és állítják elő az adatsorozatot. Leírtuk, hogy az input protokoll wrapper működése során hogyan illeszti be a lokális helyreállítható típusú adatokat ebbe az adatsorozatba, hogy a kifejezés argumentumlistáját előállíthassa. Ez által bemutattuk, hogyan rejti el a háttérben futó elosztott működést támogató kommunikációt a kifejezés elől a D-Box nyelvből generált kód.

4. fejezet

A futtató rendszer specifikációja

A csatornaindító kifejezések (auto, fix, null, startGraph, stb) futás közben csatornákat generálnak. A tényleges háttértevékenységet a futtató rendszer végzi, amely a képzendő csatornákkal kapcsolatos információk alapján dolgozik. Ezek alapján vagy egy már futó (korábban elindított) csatornát talál, vagy új csatornaindítási kérelmet azonosít.

A csatornaindító kifejezések eltérő paraméterezéssel működnek, melyekben helyet kap a csatorna típusa, és valamilyen formában annak azonosítója. Ez az azonosító lehet dinamikus, ekkor az egyedi azonosítót a futtató rendszer csak a projekt futása közben képezi.

A futtató rendszer működése során arra alapoz, hogy egy konkrét projektazonosító a futási időben is egyedi (nem beszélünk tehát projektpéldányokról). Kivétel nélkül minden csatorna előbb-utóbb egyedi azonosítót kap. A dobozok ezt az azonosítót (csatorna id) használhatják direkt hivatkozás gyanánt. Ugyanakkor a dobozok hivatkozhatnak egymás csatornáira más információkkal (indirekt) módon is. Egy csatornát szintén egyedi módon azonosítanak az alábbi információk:

- a doboz amelynél input csatornaként szerepel,
- az input szerepkörű csatornák sorrendi azonosítója (0-tól induló sorszám),
- az input szerepkörű csatornák kollekcióazonosítója (0-tól induló sorszám),

vagy

- a doboz amelynél output csatornaként szerepel,
- az output szerepkörű csatornák sorrendi azonosítója (0-tól induló sorszám).

4.1. Csatorna és doboz példányokat leíró rekordok

4.1. definíció (FUTÁSI CSATORNA LEÍRÓ). A $Cs(B_i, P_i, N_i, B_o, N_i, T, id)$ formális hetest *futási csatornaleíró* rekordnak nevezzük, ahol

- $B_i \in \{D_{box}, null\}$ az a doboz amelynél input szerepkört tölt be (vagy null),
- $P_i \in \mathbb{N}$ az input kollekció azonosító sorszáma,

- $N_i \in \mathbb{N}$ az input szerepkörön belüli sorszáma,
- $B_o \in \{D_{box}, null\}$ az a doboz amelynél output szerepkört tölt be (vagy null),
- $N_o \in \mathbb{N}$ az output szerepkörön belüli sorszáma,
- $T \in \tau$ a csatorna típusa,
- $id \in \mathbb{N}$ a csatorna generált egyedi azonosítója.

A $Cs(B_i, P_i, N_i, B_o, N_o, T, id)$ hetest egyértelmű helyzetben Cs jelöléssel fogjuk jelölni.

4.1. megjegyzés. A P_i kollekció azonosító az *autoConnBox* miatt szükséges. Ezen csatornaindító kifejezés több kollekciót indít el a csatornákból (ahány példányban a hozzá tartozó *startGraph* algráfot indított). A különböző input csatornasorozatok másmás kollekció azonosítót kapnak. Az algráfból visszafelé hivatkozó *connBox* csatlakozási kérelmek kiszolgálásakor minden algráf adott kollekciójú csatornákat kap meg, hogy adott algráfbeli kimenő csatornák ne keveredjenek össze egymással. A köztes réteg szerint egy csatorna azonosításához vagy az (id, T) párost, vagy a (B_i, P_i, N_i, T) négyest, vagy a (B_o, N_o, T) hármast kell megadni (ismerni).

4.2. definíció (BOX PÉLDÁNY). A

$$D'_{box}(threadID, box, starterThread, startedGraphs, collectNo)$$

formális ötöst *D-Box nyelvi példánynak* nevezünk, ahol

- $box \in D_{box}$ a D-Box definíció,
- $threadID \in \mathbb{N}$ a szál azonosítója,
- $starterThread \in \mathbb{N}$ az indító szál azonosítója,
- $startedGraphs \in \mathbb{N}$ a doboz által *startGraph*-al indított algráfok száma,
- $collectNo \in \mathbb{N}$ a doboz által még ki nem osztott kollekcióazonosító.

A D'_{box}(*threadID, box, starterThread, startedGraphs, collectNo*) ötöst egyértelmű helyzetben D'_{box} módon fogjuk jelölni.

4.2. megjegyzés. A doboznak van D-Box definíció formájú alakja

$$D(BoxID, subGraphID, InpProt, ExpressionDef, OutProt),$$

és van egy futási alakja

$$D'(threadID, box, starterThread, startedGraphs, collectNo).$$

A futási alakot a doboz indító függvények hozzák létre egy D-Box formájú alak példányosítása által.

4.2. A futtató rendszer állapota

A futtató rendszer minden projekthez nyilvántartja a D-Box nyelvi definíciók, a létrehozott dobozpéldányok, csatornapéldányok listáját, valamint a projekt leíró rekordok utolsó állapotát. A projekt indulásakor még nincsenek sem doboz, sem csatorna példányok, a leíró rekord mezői pedig 0 értékűek.

A projekt indítása során az 1-es algráfú dobozok példányosítása történik meg. A dobozpéldányok ezek után elkezdik a saját input és output csatornáikat példányosítani, illetve a példányosított csatornákra rákeresnek. Ezek a folyamatok a valóságban időben párhuzamosan zajlanak, de a csatornaindító függvények lefutása időbe kerül. Különösen problémás a *startGraph* - *autoConnBox* párosok végrehajtása. A *autoConnBox* kifejezés végrehajtását meg kell előzze a hozzá tartozó *startGraph* lefutása. A *startGraph*-ok azonban csak akkor tudnak befejeződni, ha az általuk indított algráf bevezető dobozainak input csatornái létrejöttek, hisz a *startGraph* csatornák ide fognak csatlakozni. Az algráf dobozainak csatornái azonban nem tudnak indulni, mivel az utolsó doboz output csatornái az *autoConnBox* által generált input csatornákra akarnak felcsatlakozni.

A valós életben a dobozok indulásuk után önállóan működnek, elkezdik saját input és output csatornáikat felépíteni. Mivel a dobozok fizikailag különböző gépeken indulnak el, ezek a folyamatok azonos időben történnek. A fenti *startGraph* - *autoConnBox* probléma orvosolható oly módon, hogy az *autoConnBox* megvárja a *startGraph* általi indítást, és az algráf dobozok is megvárják amíg az *autoConnBox* felépíti az input csatornáit.

A futó projektben párhuzamosan zajló folyamatok bővítik, módosítják a futó dobozok, és a csatornák listáját. A projekt állapotának leírását központosított módon a futtató rendszer tárolja és kezeli. Figyelembe kell venni, hogy egyes műveletek végrehajtása atomi kell legyen, például új szálazonosító generálása, fix csatorna indítása, stb.

Implementáció során bizonyos műveletek atomitását valamiféle *monitor* biztosítja. Jelen specifikáció azonban ezt nem használja ki, egy időben csak egy köztes réteg állapotát módosító folyamat lehet aktív.

4.3. definíció (KÉSLELTETETT CONN). A $D_{spec} = (boxid, thread, types)$ formális hármast késleltetett csatorna leírónak nevezzük, ahol $boxid \in String$ doboz azonosító, $thread \in \mathbb{N}$ egy szál azonosítója, $types \in \langle T_\tau \rangle$, $types = \langle t_0, t_1, \ldots, t_{n-1} \rangle$ típusok sorozata.

4.3. megjegyzés. A D_{spec} speciális leíró abból a szempontból, hogy a szálazonosító egy konkrét, futás közbeni szál azonosítója, nem pedig (thisThread, ancestorThread) relatív hivatkozás. A *startGraph* output csatorna indításakor fogjuk hasznát venni

ennek a csatorna leírónak.

4.4. definíció (CSATORNA INDÍTÁS). A $D_c(B, IO, D)$ formális hármast csatornaindítás leírásnak nevezzük, ahol

- $B \in D_{box}$ a doboz példány, akihez a csatornaindítás tartozik,
- $IO \in \{input, output\}$ a csatorna jellege,
- $D \in \{D_{null}, D_{fix}, D_{auto}, D_{conn}, D_{connBox}, D_{startGraph}, D_{spec}\}$ a csatornaindítás paraméterei,
- $step \in \mathbb{N}$ a csatornaindítás sorszáma.

4.4. megjegyzés. Amikor egy doboz példányosításra kerül, akkor a dobozban szereplő input és output csatornák indításai bekerülnek a futtató rendszer állapotába mint végrehajtandó *utasítások*. Az indítások sorszámozva vannak. Egy n sorszámú csatorna indítást nem szabad addig végrehajtani, amig a $[0, n-1]$ sorszámú csatorna indítások mindegyike be nem fejeződött. A sorszámozás és az indítási feltételek külön kezelődnek input és output csatorna indítások esetén.

4.5. definíció (FUTTATÓ RENDSZER ÁLLAPOTA). A $MW(D, D', C', lt, lc)$ formális hatost *futtató rendszer állapotnak* nevezzük, ahol

- $D \in \langle D_{box} \rangle$ D-Box definíciók sorozata,
- $D' \in \langle D'_{box} \rangle$ D-Box példányok,
- $C' \in \langle D \rangle$ csatorna-példányok,
- $C_s \in \langle D_c \rangle$ a még nem végrehajtott input/output csatorna indítások,
- $lt \in \mathbb{N}$ a futás során utoljára kiosztott szál-azonosító,
- $lc \in \mathbb{N}$ a futás során utoljára kiosztott csatorna-azonosító.

4.5. megjegyzés. Egy futtató rendszer állapot ismeri (tárolja) a projektet alkotó D-Box definíciókat, és a futás közbeni példányokat. Ezen felül az egyedi sorszámkiosztás támogatásához leírással rendelkezik az utoljára kiosztott szál és csatorna azonosítókról. A projekt futása jellemezhető a futtató rendszer állapotok valamely sorozatával, ahol a sorozat egymást követő elemeit műveletek (állapottranszformációs függvények) alakítják. A sorozat egymást követő elemei az időbeni sorrendiséget tükrözik.

4.6. definíció (KEZDETI ÁLLAPOT). Egy MW futtató rendszer *kezdeti állapotának* nevezzük, amikor

- $MW.D = \langle d_0, d_1, \ldots, d_{n-1} \rangle$ D-Box definíciók adottak,
- $MW.D' = \emptyset$ még nincs futó dobozpéldány,
- $MW.C' = \emptyset$ még nincs futó csatornapéldány,
- $MW.C_s = \emptyset$ nincsenek várakozó csatornaindítások,
- $MW.lt = 0$ szál-azonosító még nem került kiosztásra,

- $MW.lc =$ a projektben szereplő fix csatornaazonosítók legnagyobb értékét tartalmazza.

A továbbiakban S_0-al jelöljük a futtató rendszer ezen kezdeti állapotát.

4.7. definíció (DOBOZPÉLDÁNY KIKERES). Legyen $f_{findB}\colon MW \times \mathbb{N} \times String \to \{D', null\}$, $f_{findB}(M, thr, box) = d_r$ függvény, ahol ha $M.D' = \langle d_0, d_1, \ldots, d_{n-1}\rangle$, $H \coloneqq \{b\colon b \in M.D' \wedge b.threadID = thr \wedge b.boxID = box\}$, és

- ha $sizeof(H) = 1, H = \{b_0\}$, akkor $d_r \coloneqq b_0$,
- különben $d_r \coloneqq null$.

4.6. megjegyzés. A f_{findB} függvény meghatározza (kikeresi) az adott futtató rendszerbeli állapotban szereplő dobozpéldányokból azt a dobozpéldányt, amely adott szálazonosítóval és $boxID$-vel rendelkezik.

4.8. definíció (CSATORNAPÉLDÁNY KIKERES). Legyen $f_{findC}\colon MW \times T_\tau \times \mathbb{N} \to \{Cs, null\}$, $f_{findC}(M, t, id) = c_r$ függvény, ahol $M.C' = \langle c_0, c_1, \ldots, c_{n-1}\rangle$, és $H \coloneqq \coloneqq \{c\colon c \in M.C' \wedge c.t = t \wedge c.id = id\}$,

- ha $sizeof(H) = 1, H = \{c_0\}$, akkor $c_r \coloneqq c_0$,
- különben $c_r \coloneqq null$.

4.7. megjegyzés. A f_{findC} függvény meghatározza (kikeresi) az adott futtató rendszerbeli állapotban szereplő csatornapéldányokból azt a csatornapéldányt, amely adott típussal és id-val rendelkezik.

4.9. definíció (DOBOZ CSATORNA KIKERES). Legyen $f_{findBC}\colon MW \times T_\tau \times String \times \times \mathbb{N} \times \mathbb{N} \to \{Cs, null\}$, $f_{findCB}(M, t, d, coll, n) = c_r$ olyan függvény, ahol ha az aktuális csatornapéldányok $M.C' = \langle c_0, c_1, \ldots, c_{n-1}\rangle$, és $H \coloneqq \{c\colon c \in M.C' \wedge c.t = t \wedge \wedge c.B_i.boxID = d \wedge c.P_i = coll \wedge c.N_i = n\}$ esetén

- ha $sizeof(H) = 1, H = \{c_0\}$, akkor $c_r \coloneqq c_0$,
- különben $c_r \coloneqq null$.

4.8. megjegyzés. A f_{findBC} függvény adott dobozazonosító (d) adott kollekciójába ($coll$) tartozó, adott sorszámú (n) input csatornáját keresi ki.

4.10. definíció (INPUT SORREND MAXIMUMA). Legyen $f_{maxInp}\colon \langle Cs\rangle \times D'_{box} \times \mathbb{N} \to \mathbb{N}$, $f_{maxInp}(C_b, D, coll) = m$ függvény, és $C \coloneqq \{c.N_i\colon c \in C_b, c.B_i.BoxID = D.boxID \wedge \wedge c.P_i = coll\}$,

- ha $sizeof(C) = 0$ akkor $m \coloneqq 0$,
- ha $sizeof(C) > 0$ akkor $m \coloneqq max(C)$.

4.9. megjegyzés. Az f_{maxInp} függvény megkeresi egy csatornasorozatban szereplő csatornák közül azokat, amelyek adott doboz (D) adott kollekciójában ($coll$) input csatornaként vannak regisztrálva, majd ezen csatornák (N_i) input csatorna sorrendi értékei közül megadja a legnagyobbat. Ha nincsenek ilyen csatornák, akkor 0-t ad meg.

4.11. definíció (OUTPUT SORREND MAXIMUMA). Legyen az $f_{maxOut}\colon MW \times \mathbb{N} \times \times String \to \mathbb{N}$, $f_{maxOut}(M, thr, box) = m$ függvény, és $C \coloneqq \{c.N_o\colon c \in MW.C' \wedge \wedge c.B_o.threadID = thr \wedge c.B_o.boxID = box\}$,

- ha $sizeof(C) = 0$ akkor $m \coloneqq 0$
- ha $sizeof(C) > 0$ akkor $m \coloneqq max(C)$.

4.10. megjegyzés. Ez a függvény hasonlóan az f_{maxInp} függvényhez, megkeresi egy csatornasorozatban szereplő csatornák közül azokat, amelyek adott doboz output csatornáiként vannak regisztrálva, majd ezen csatornák N_o, output csatorna sorrendi értékei közül megadja a legnagyobbat. Ha nincsenek ilyen csatornák, akkor 0-t ad meg.

4.12. definíció (CSATORNAÁLLAPOT FRISSÍTÉSE). Legyen $f_{\odot}\colon \langle Cs\rangle \times Cs \to \langle Cs\rangle$ függvény, ahol $f_{\odot}(C_b, c_r) = C'_b$, és $C_b = \langle c_0, c_1, \ldots, c_{n-1}\rangle$, $H \coloneqq \{i\colon c_i.T = c_r.T \wedge c_i.id = = c_r.id \wedge i \in [0, n-1]\}$,

- ha $sizeof(C) = 0$, akkor $C'_b \coloneqq C'_b \oplus c_r$,
- ha $sizeof(C) = 1$, akkor $j \coloneqq min(H)$, $C'_b \coloneqq \langle c_0, c_1, \ldots, c_{j-1}, c_r, c_{j+1}, \ldots, c_{n-1}\rangle$.

Ezt a függvényt a továbbiakban operátor alakban fogjuk használni: $C'_b \coloneqq C_b \odot c_r$.

4.11. megjegyzés. Az $\odot$ művelet feltételes alakja a korábban definiált $\oplus$ műveletnek. Amennyiben a csatornák sorozatában az adott típusú és csatornaazonosítójú csatornapéldány már szerepelne, akkor azt kicseréli az új példányra. Ellenkező esetben hozzáfűzi azt a sorozat végére.

4.13. definíció (DOBOZÁLLAPOT FRISSÍTÉS). Legyen $f_{\odot}\colon \langle D'_{box}\rangle \times D'_{box} \to \langle D'_{box}\rangle$ függvény, ahol $f_{\odot}(D_b, d_r) = D'_b$, és $D_b = \langle d_0, d_1, \ldots, d_{n-1}\rangle$, $H \coloneqq \{i\colon d_i.threadID = = d_r.threadID \wedge d_i.boxID = d_r.boxID \wedge i \in [0, n-1]\}$,

- ha $sizeof(H) = 0$, akkor $D'_b \coloneqq D'_b \oplus d_r$,
- ha $sizeof(H) = 1$, akkor $j \coloneqq min(H)$, $D'_b \coloneqq \langle d_0, d_1, \ldots, d_{j-1}, d_r, d_{j+1}, \ldots, d_{n-1}\rangle$.

Ezt a függvényt a továbbiakban operátor alakban fogjuk használni: $D'_b \coloneqq D_b \odot d_r$.

4.12. megjegyzés. A $\odot$ művelet hasonlóan a csatornákon értelmezett esethez, az adott dobozt vagy hozzáadja a sorozathoz, vagy ha ugyanilyen szálazonosítójú és dobozazonosítójú példány már létezett korábban a sorozatban, akkor azt az új állapotú példányra cseréli le.

4.3. Futtató rendszer szemantika

A szemantika leírását *természetes műveletei szemantika* szerint adjuk meg. Ennek során $s \in MW$ futtató rendszer állapotról $s' \in MW$ állapotra képezünk le. Az induló állapot a 4.6. definíció szerinti $s \coloneqq S_0$ állapot lesz.

Műveletek, melyek szemantikai leírását megfogalmazzuk a továbbiakban:

- doboz indítása,
- algráfba tartozó összes doboz indítása,
- az 1-es algráf dobozainak indítása,
- input csatornák indítása,
- output csatornák indítása.

4.14. definíció (új szálazonosító). A *nthr* művelet egy új szálazonosítót képez a futtató rendszer állapotában. Szemantikája:

$$[nthr]\frac{s' \coloneqq s[lt \mapsto \mathcal{N}[\![lt+1]\!]s]}{\langle nthr, s\rangle \to s'}$$

4.13. megjegyzés. A művelet azt a futtató rendszer állapotot definiálja, amelyben az utoljára kiosztott szálazonosító értéke növekedett 1-gyel. Az implementáció során ügyelni kell arra, hogy ez a műveletvégrehajtás atomi lépés legyen.

4.15. definíció (új csatorna azonosító). A *nch* művelet egy új csatorna azonosítót képez a futtató rendszer állapotában. Szemantikája:

$$[nch]\frac{s' \coloneqq s[lc \mapsto \mathcal{N}[\![lc+1]\!]s]}{\langle nch, s\rangle \to s'}$$

4.14. megjegyzés. A művelet azt a futtató rendszer állapotot definiálja, amelyben az utoljára kiosztott csatornaazonosító értéke növekedett 1-gyel. Az implementáció során ügyelni kell arra, hogy ez a műveletvégrehajtás atomi lépés legyen.

4.16. definíció (startBox művelet). A *sb* doboz indító művelet paramétere egy $thr \in \in \mathbb{N}$ a dobozt indító szálazonosító (ancestorThread), valamint egy $boxID \in String$ doboz azonosító. Szemantikája:

$$[sb]\frac{s' \coloneqq s[D' \mapsto D' \oplus d', C' \mapsto C' \oplus C_i \oplus C_o]}{\langle sb(thr, boxID), s\rangle \to s'}$$

ahol

- $d' \coloneqq D'_{box}(s.lt, boxID, thr, 0, none, none)$ képviseli az új, elindított doboz példányt,
- ha $d'.InpProt.IChannels = \langle (f_0, d_0), (f_1, d_1), \ldots, (f_{n-1}, d_{n-1})\rangle$,
- $C_i \coloneqq \langle S_0, S_1, \ldots, S_{n-1}\rangle$ ahol $\forall i \in [0, n-1]$ esetén $S_i \coloneqq D_c(d', input, s_i, i)$ az i. input csatorna indító leiró,
- ha $d'.OutProt.OChannels = \langle (f_0, d_0), (f_1, d_1), \ldots, (f_{n-1}, d_{m-1})\rangle$,
- $C_o \coloneqq \langle P_0, P_1, \ldots, P_{m-1}\rangle$ ahol $\forall i \in [0, m-1]$ esetén $P_i \coloneqq D_c(d', output, p_i, i)$ az i. output csatorna indító leiró.

4.15. megjegyzés. A *startBox* művelet példányosít egy konkrét D-Box definíciót, és beilleszti a példányt a futtató rendszer példányokat nyilvántartó sorozatába. A művelet első paramétere az indító szál azonosítója, a második az indítandó doboz ID-je. A következőkben a futtató rendszer állapotába illeszti a saját input és output csatorna indítási kérelmeit. Az implementáció során a kódkészletből az adott D-Box kódot egy adott *node*-on el kell indítani (példányosítani), és a dobozt inicializálni kell.

4.17. definíció (ALGRÁF INDÍTÁSA). Az *sg* művelet paramétere egy $t \in \mathbb{N}$ a dobozt indító szálazonosító (ancestorThread), valamint $g \in \mathbb{N}$ az indítandó algráf azonosító. A művelet az összes dobozt indítja az adott algráfból. Szemantikája:

$$[sg]\frac{\langle nthr, s\rangle \to s_0, \langle sb(t, b_0), s_0\rangle \to s_1, \langle sb(t, b_1), s_1\rangle \to s_2, \ldots, \langle sb(t, b_{n-1}), s_{n-1}\rangle \to s_n}{\langle sbs(t, g), s\rangle \to s_n}$$

ahol $boxes = \langle b.boxID \colon b \in s.D, d.subGraphId = g\rangle$, $boxes = \langle b_0, b_1, \ldots, b_{n-1}\rangle$.

4.18. definíció (STARTGRAPH N MŰV). Az *sgn* művelet paramétere egy $t \in \mathbb{N}$ a dobozt indító szálazonosító (ancestorThread), $g \in \mathbb{N}$ az indítandó algráf azonosító, valamint $n \in \mathbb{N}$ az indítási darabszám. A művelet az adott algráfot n-szer indítja el. Szemantikája:

$$[sgn]\frac{\langle sg(t, g), s_0\rangle \to s_1, \langle sg(t, g), s_1\rangle \to s_2, \ldots, \langle sg(t, g), s_{n-1}\rangle \to s_n}{\langle sgn(t, g, n), s_0\rangle \to s_n}$$

4.19. definíció (MAIN MŰV). A *main* művelet az 1-es algráfot indítja el 1 példányban. Az indítás feltétele, hogy a futtató rendszer kezdőállapotban legyen.

$$[start]\ \frac{\langle sgn(0,1,1), s\rangle \to s'}{\langle main, s\rangle \to s'} \text{ ha } s = S_0.$$

4.16. megjegyzés. A *main* művelet használható a projekt indítására. Eredményül adja azt az állapotot, ahol az 1-es algráfba tartozó dobozok mindegyike példányosításra került. A *main* műveletet csak a futtató rendszer S_0 állapotán lehet elvégezni.

4.4. Csatornaindítások

A csatornaindítások szemantikájának lényege, hogy a futtató rendszer állapota egy új csatornapéldánnyal bővül, és a végrehajtott csatornaindító utasítással csökken az utasításpuffer. A csatornapéldányt be kell illeszteni az adott doboz input vagy output csatornáinak listájába, inicializálni kell, és el kell látni sorszámmal.

4.20. definíció (INPUT CSATORNA START). Az sch_i input csatornaindító művelet paramétere egy $C \in D_c$ csatornaindítási leírás, egy id csatornaazonosító, és egy pid kollekcióazonosító. Szemantikája:

$$[sch_i]\frac{s' \coloneqq s[C' \mapsto C' \oplus c_{new}]}{\langle sch_i(C, id, pid), s\rangle \rightarrow s'}$$

ahol

- $o \coloneqq f_{maxInp}(s.C', C.B) + 1$ a csatorna dobozon belüli input sorszáma,
- $c_{new} \coloneqq Cs(C.B.boxID, pid, o, null, -1, C.type, id)$ az új csatornapéldány,
- a c_{new} példányt az $init()$ művelettel inicializálni kell.

4.17. megjegyzés. Az sch_i művelet egy konkrét típusú és azonosítójú csatornapéldányt készít el, és illeszt a csatornák listájára. Az implementáció során a kódkészletből adott *type* típusú csatorna kódot egy adott *node*-on el kell indítani (példányosítani), és a csatornát az adott *id* azonosítóra kell inicializálni.

4.21. definíció (OUTPUT CSATORNA START). Az sch_o output csatornaindító művelet paramétere egy $C \in D_c$ csatornaindítási leírás, és egy id csatornaazonosító. Szemantikája:

$$[sch_o]\frac{s' \coloneqq s[C' \mapsto C' \oplus c_{new}]}{\langle sch_o(C, id), s\rangle \rightarrow s'}$$

ahol

- $o \coloneqq f_{maxOut}(s.C', C.B) + 1$ a csatorna dobozon belüli output sorszáma,
- $c_{new} \coloneqq Cs(null, -1, -1, C.B.boxID, o, C.type, id)$ az új csatornapéldány,
- melyet az $init()$ műveletével inicizalizálni kell.

4.18. megjegyzés. Az sch_o művelet egy konkrét típusú és azonosítójú csatornapéldányt készít el, és illeszt a csatornák listájára. Az implementáció során a kódkészletből adott *type* típusú csatorna kódot egy adott *node*-on el kell indítani (példányosítani), és a csatornát az adott *id* azonosítóra kell inicializálni.

4.22. definíció (OUTPUT SET). A set_o művelet paramétere egy beállítás szempontjából frissítendő csatorna ($C \in Cs$), és egy $B \in D_{box}$ doboz példány, amely outputként kívánja ezen csatornát használni. Szemantikája:

$$[set_o]\frac{s' \coloneqq s[C' \mapsto C' \odot cs]}{\langle set_o(C,B), s\rangle \to s'}$$

ahol $n \coloneqq f_{maxOut}(s, B.threadID, B.boxID) + 1$ a következő output csatorna sorszáma a doboznak, $cs \coloneqq C[B_o \mapsto B, N_o \mapsto n]$ a frissített csatorna rekord.

4.23. definíció (REMOVE). A *rem* művelet paramétere $C \in D_C$ csatornaindítást leíró rekord. A művelet eltávolítja ezt a leírót a futtató rendszer állapotából. Szemantikája:

$$[rem]\frac{s' \coloneqq s[C_s \mapsto C_s \ominus C]}{\langle rem(C), s\rangle \to s'}$$

4.5. Az input csatornaindítások

4.24. definíció (INPUT NULL). A $null_i$ művelet paramétere egy $C \in D_c$ input csatornaindítási leíró, ahol $C.D \simeq D_{null}$. Szemantikája:

$$[null_i]\ \frac{\langle rem(C), s\rangle \to s'}{\langle null_i(C), s\rangle \to s'} \text{ ha } C.IO = input \wedge C.D \simeq D_{null}$$

4.25. definíció (INPUT AUTO). Az $auto_i$ művelet paramétere egy $C \in D_c$ input csatornaindítási leíró, ahol $C.D \simeq D_{auto}$. Szemantikája:

$$[auto_i]\ \frac{\begin{array}{c}\langle nch, s\rangle \to s_0, \langle sch_i(C, s_0.lc, 0), s_0\rangle \to s_1,\\ \langle rem(C), s_1\rangle \to s'\end{array}}{\langle auto_i(C), s\rangle \to s'} \text{ ha } C.IO = input \wedge C.D \simeq D_{auto}$$

4.26. definíció (INPUT FIX). A fix_i művelet paramétere egy $C \in D_c$ input csatornaindítási leíró, ahol $C.D \simeq D_{fix}$. Szemantikája:

$$[fix_i]\ \frac{\langle sch_i(C, C.D.id, 0), s\rangle \to s_0, \langle rem(C), s_0\rangle \to s'}{\langle fix_i(C), s\rangle \to s'} \text{ ha } C.IO = input \wedge C.D \simeq D_{fix}$$

4.27. definíció (INPUT $conn_s$). A $conn_s$ művelet paramétere egy $B \in D_{box}$ doboz, amely az indítást hajtja végre, egy $t \in T_\tau$ típus, és egy *pid* kollekcióazonosító. Szemantikája:

$$[conn_s]\frac{\langle nch, s\rangle \to s_0, \langle sch_i(C, s_0.lc, pid), s\rangle \to s'}{\langle s_{conn}(B, t, pid), s\rangle \to s'}$$

ahol $C \coloneqq D_c(B, input, D_{auto}(t, auto))$.

4.19. megjegyzés. A $conn_s$ művelet generál egy új csatornaazonosítót, majd indít egy új input csatornapéldányt.

4.28. definíció (INPUT i_{conn}). Az $conn_p$ művelet paramétere egy $C \in D_c$ input csatornaindítási leíró, ahol $C.D \simeq D_{conn}$, valamint egy *pid* kollekcióazonosító. Szemantikája:

$$[conn_p]\ \frac{\begin{array}{c}\langle conn_s(C.B, t_0, pid), s\rangle \to s_1,\\ \langle conn_s(C.B, t_1, pid), s_1\rangle \to s_2, \ldots,\\ \langle conn_s(C.B, t_{n-1}, pid), s_{n-1}\rangle \to s_n\end{array}}{\langle conn_p(C, pid), s\rangle \to s_n} \text{ ha } C.IO = input \wedge C.D \simeq D_{conn}$$

ahol $C.D.types = \langle t_0, t_1, \ldots, t_{n-1}\rangle$ a csatornák típusai.

4.20. megjegyzés. Az $conn_p$ művelet az $conn_i$ művelet segédlépése. Egy kollekciónyi input csatornát indít el, melyeknek típusai a D_{conn} leíró szerintiek.

4.29. definíció (INPUT *Iconn*). A $conn_i$ művelet paramétere egy D_{conn} input csatornaindítási leíró, ahol $C.D \simeq D_{conn}$. Szemantikája:

$$[conn_i]\ \frac{\langle i_{conn}(C,1), s\rangle \to s_1, \langle i_{conn}(C,2), s_1\rangle \to s_2, \ldots, \langle i_{conn}(C,n), s_{n-1}\rangle \to s_n}{\langle conn_i(C), s\rangle \to s_n} \text{ ha } n > 0$$

ahol $n = C.B.startedGraphs$.

4.21. megjegyzés. A $conn_i$ művelet csak akkor hajtható végre, ha a D_{conn} leíróban hivatkozott, a *startGraph* műveletet tartalmazó doboz már befejezte a műveletet (ekkor a *startedGraph* értéke ezen dobozpéldányban nagyobb lesz mint 0.

4.6. Az output csatornaindítások

4.30. definíció (OUTPUT NULL). Az $null_o$ művelet paramétere egy $C \in D_c$ output csatornaindítási leíró, ahol $C.D \simeq D_{null}$. Szemantikája:

$$[null_o]\ \frac{\langle rem(C), s\rangle \to s'}{\langle null_o(C), s\rangle \to s'} \text{ ha } C.IO = output \wedge C.D \simeq D_{null}$$

4.31. definíció (OUTPUT FIX). A fix_o művelet paramétere egy $C \in D_C$ output csatornaindítási leíró, ahol $C.D \simeq D_{fix}$. Szemantikája:

$$[fix_o]\ \frac{\begin{array}{c}\langle set_o(C_c), s\rangle \to s_m,\\ \langle rem(C), s_m\rangle \to s'\end{array}}{\langle Ofix(C), s\rangle \to s'} \text{ ha } C.IO = output \wedge C.D \simeq D_{fix} \wedge C_c \neq null$$

ahol $C_c \coloneqq f_{findC}(s, C.D.type, C.D.id)$ az adott típusú és azonosítójú csatornapéldány.

4.22. megjegyzés. A fix_o valójában nem indít csatornát. Akkor alkalmazható, ha az adott *id*-vel rendelkező fix csatorna az állapot szerint már indításra került (input csatornaként). Ekkor a csatornapéldány beállításait módosítani kell, hogy bekerüljön, mely doboz használja output célokra.

4.32. definíció (OUTPUT CONNBOX THISTHREAD). Az cb_t művelet paramétere egy $D_{connBox}thisThread$ output csatornaindítási leíró. Szemantikája:

$$[cb_t]\ \frac{\langle set_o(C_0,b),s_0\rangle \to s_1, \langle set_o(C_1,b),s_1\rangle \to s_2,\ldots,\ \langle set_o(C_{n-1},b),s_{n-1}\rangle \to s_n, \langle rem(C),s_n\rangle \to s'}{\langle cb_a(C),s\rangle \to s'}\ \text{ha}\ \begin{cases} C.IO = output \wedge \\ C.D \simeq D_{connBox} \wedge \\ C.D.thr = thisThread \wedge \\ null \notin C_c \end{cases}$$

ahol

- $s_0 \coloneqq s$,
- $B_0 \coloneqq f_{findB}(s, C.D.threadID, C.D.boxID)$ az indító doboz,
- $b \coloneqq B_0.boxID$ az azonosítója,
- $C.D.types = \langle t_0, t_1, \ldots, t_{n-1}\rangle$ a csatornák típusai,
- ekkor $C_c \coloneqq \langle C_0, C_1, \ldots, C_{n-1}\rangle$ a csatornaindítási leírók,
- ahol $B \coloneqq f_{findB}(s, B_0.starterThread, C.D.boxID)$ a hivatkozott doboz,
- $\forall i \in [0, n-1]$ esetén $C_i \coloneqq f_{findBC}(s_i, B.threadID, B.boxID, 0, i)$.

4.33. definíció (OUTPUT CONNBOX ANCESTORTHREAD). Az cb_a művelet paramétere egy $D_{connBoxancestorThread}$ output csatornaindítási leíró. Szemantikája:

$$[cb_a]\ \frac{\langle ncoll(B),s\rangle \to s_0, \langle set_o(C_0,b),s_0\rangle \to s_1,\ \langle set_o(C_1,b),s_1\rangle \to s_2,\ldots,\ \langle set_o(C_{n-1},b),s_{n-1}\rangle \to s_n, \langle rem(C),s_n\rangle \to s'}{\langle cb_a(C),s\rangle \to s'}\ \text{ha}\ \begin{cases} C.IO = output \wedge \\ C.D \simeq D_{connBox} \wedge \\ C.D.thr = ancestorThread \wedge \\ null \notin C_c \end{cases}$$

ahol

- $B_0 \coloneqq f_{findB}(s, C.D.threadID, C.D.boxID)$ az indító doboz,
- $b \coloneqq B_0.boxID$ az azonosítója,
- $C.D.types = \langle t_0, t_1, \ldots, t_{n-1}\rangle$ a csatornák típusai,
- $B \coloneqq f_{findB}(s_0, B_0.starterThread, C.D.boxID)$ a hivatkozott doboz,
- $C_c \coloneqq \langle C_0, C_1, \ldots, C_{n-1}\rangle$ a csatornaindítási leírók,
  $\forall i \in [0, n-1]$ esetén $C_i \coloneqq f_{findBC}(s_i, B.threadID, B.boxID, B.collectNo, i)$.

4.34. definíció (OUTPUT SPEC). Az $spec_o$ művelet paramétere egy $C \in D_c$ output csatornaindítási leíró, ahol $C.D \simeq D_{spec}$. Szemantikája:

$$[spec_o]\ \frac{\langle set_o(C_0,C.B),s_0\rangle \to s_1,\ \langle set_o(C_1,C.B),s_1\rangle \to s_2,\ldots,\ \langle set_o(C_{n-1},C.B),s_{n-1}\rangle \to s_n}{\langle spec_o(C),s\rangle \to s'}\ \text{ha}\ C.IO = output \wedge C.D \simeq D_{spec} \wedge null \notin C_c,$$

ahol $C.D.types = \langle t_0, t_1, \ldots, t_{n-1}\rangle$, $C_c \coloneqq \langle C_0, C_1, \ldots C_{n-1}\rangle$, $\forall i \in [0, n-1]$ esetén $C_i \coloneqq f_{findBC}(s_i, t_i, C.D.boxid, C.D.thread, 0, i)$.

4.23. megjegyzés. A $spec_o$ művelet adott $C.D.thread$ szál $C.D.boxid$ doboz input csatornáira csatlakozik fel, amelyek típusát és számát a $C.D.types$ írja le. A keresett csatornák *auto* jellegűek, mindegyikük 0-s kollekcióba tartozik. A művelet sikerességéhez szükséges, hogy a cél doboz korábban indítsa el ezeket az input csatornákat.

4.35. definíció (addN). Az $addN$ művelet paramétere $C_c \in \langle D_c \rangle$ csatornaindítás leírók sorozata, melyeket a futtató rendszer állapotába kell illeszteni. Szemantikája:

$$[addN]\frac{s' \coloneqq s[C_s \mapsto C_s \oplus c_0 \oplus c_1 \oplus \cdots \oplus c_{n-1}]}{\langle addN(C_c), s\rangle \to s'}$$

ahol $C_c = \langle c_0, c_1, \ldots, c_{n-1}\rangle$.

4.36. definíció (output startGraph). Az sg_o művelet paramétere egy $C \in D_c$ output csatornaindítási leíró, ahol $C.D \simeq D_{startGraph}$. Szemantikája:

$$[sg_o]\ \frac{\begin{array}{c}\langle sgn(t,g,n), s\rangle \to s_0,\\ \langle addN(C_s), s_0\rangle \to s_1, \langle rem(C), s_1\rangle \to s'\end{array}}{\langle OstartGraph(C), s\rangle \to s'} \text{ ha } C.IO = output \wedge C.D \simeq D_{startGraph}$$

ahol

- $n \coloneqq \mathcal{A}[\![C.D.count]\!]s$ indítási darabszám,
- $t \coloneqq C.B.threadID$ az indító doboz szálazonosítója,
- $g \coloneqq C.D.sid$ az indítandó algráf azonosítója,
- $m \coloneqq s.lt$ az utolsó indított szál azonosító
- $C_s \coloneqq \langle C_0, C_1, \ldots, C_{n-1}\rangle$ a specilis csatornaindítási leírók, $\forall i \in [0, n-1]$ esetén $C_i \coloneqq D_{spec}(C.D.boxid, m+i+1, C.D.types, C.step)$.

4.24. megjegyzés. A lépések az alábbiak:

- először meghatározzuk az indítási darabszámot (n értékét),
- ennyiszer indítjuk el a megfelelő algráf dobozait az sgn művelet segítségével,
- elhelyezünk speciális csatlakozási leírókat az állapotban,
- majd eltávolítjuk a C végrehajtott lépést.

A problémát az okozza, hogy a sg_o által indított dobozok még nem indították el az input csatornáikat, ezért nem lehet befejezni ezen műveletet. Meg kell várni, míg az input csatornák elindulnak. Ekkor a késleltetett *spec* leírók segítségével az elindult input csatornák leíróit frissíteni fogja a futtató rendszer.

4.37. definíció (input lépés). A következő végrehajtható input csatornaindítás. Szemantikája:

$$[input]\ \frac{\langle inext(lep), s\rangle \to s'}{\langle input, s\rangle \to s'} \text{ ha } lep \neq null$$

ahol $H \coloneqq \{d\colon d \in C_s, d.IO = input, \nexists d' \in C_s, d'.B = d.B \wedge D'.step < d.step\}$ esetén ha $H = \emptyset$ akkor $lep = null$, különben lep legyen a H halmaz tetszőleges eleme. A $lep \neq null$ esetén az $inext$ valójában egy ...

- $Inull$ műveletet jelöl, ha $lep.D \simeq D_{null}$,
- $Ifix$ műveletet jelöl, ha $lep.D \simeq D_{fix}$,
- $Iauto$ műveletet jelöl, ha $lep.D \simeq D_{auto}$,
- $Iconn$ műveletet jelöl, ha $lep.D \simeq D_{conn}$.

4.25. megjegyzés. Az input lépés egy input csatornaindítást választ ki a tárolt lépéssorozatból, majd végrehajtja. A kiválasztásnál ügyelni kell arra, hogy a lépés az adott doboz következő sorszámú input csatornaindítása legyen, mivel az input csatornák sorrendi értéke csak ekkor lesz helyesen beállítva.

4.38. definíció (OUTPUT LÉPÉS). A következő végrehajtható output csatornaindítás. Szemantikája:

$$[output]\ \frac{\langle onext(lep), s\rangle \rightarrow s'}{\langle output, s\rangle \rightarrow s'} \text{ ha } lep \neq null,$$

ahol $H \coloneqq \{d\colon d \in C_s, d.IO = output, \nexists d' \in C_s, d'.B = d.B \wedge D'.step < d.step\}$ esetén ha $H = \emptyset$ akkor $lep = null$ különben lep legyen a H halmaz tetszőleges eleme. A $lep \neq null$ esetén az $onext$ valójában egy ...

- $Onull$ műveletet jelöl, ha $lep.D \simeq D_{null}$,
- $Ofix$ műveletet jelöl, ha $lep.D \simeq D_{fix}$,
- $OconnT$ műveletet jelöl, ha $lep.D \simeq D_{connBox}$, és $lep.thr = thisThread$,
- $OconnA$ műveletet jelöl, ha $lep.D \simeq D_{connBox}$, és $lep.thr = ancestorThread$,
- $OstartGraph$ műveletet jelöl, ha $lep \simeq D_{startGraph}$.

4.39. definíció (FINISH). A *finish* utasítás nem változtatja meg a futtató rendszer állapotát. Szemantikája:

$$[finish]\ \frac{s' \coloneqq s}{\langle finish, s\rangle \rightarrow s'} \text{ ha } s \neq S_0 \wedge s.C_s = \emptyset$$

4.26. megjegyzés. A *finish* művelet akkor választható ki, ha minden csatornaindítási lépés befejeződött. Ezen művelet nem változtatja meg a futtató rendszer állapotát.

A futtató rendszer programja négy utasításból áll:

```
main;
input;
output;
finish;
```

A futtató rendszer állapota induláskor S_0 kezdő állapot. Első lépésként csak a *main* hajtható végre, ennek hatására a futtató rendszer az 1-es algráfba tartozó dobozokat

indítja el. A dobozok példányai bekerülnek az $s.B'$ sorozatba, a dobozok csatornaindítási utasításai pedig az $s.C_s$ sorozatba. A *main* többé nem kiválasztható utasítás, mivel a rendszer elmozdul a kezdő állapotból.

A *input* és *output* lépések addig kerülnek kiválasztásra (valamilyen sorrendben), amíg a $s.C_s$ sorozat ki nem ürül, vagyis van indítandó csatorna. Gyakorlatilag mindegy, hogy milyen sorrendben kerülnek indításra a csatornák: a sorrend csak dobozon belül számít. Egy doboz input (és output) csatornái csak megfelelő sorrendben indíthatóak. Ezt a sorrendet az *input* és *output* lépések betartják. Amennyiben vagy a soron következő *input* vagy *output* lépés mégsem végrehajtható, úgy a futtató rendszer egy másik *input* vagy *output* indítást választ ki. Elképzelhető, hogy egy *input* vagy *output* indítás egy másik doboz *output* vagy *input* indításától függ. Ez esetben a futás során ez a lépés előbb–utóbb kiválasztásra kerül, és a korábban nem végrehajtott művelet is befejeződhet a legközelebbi kiválasztáskor.

Amennyiben az $s.C_s$ sorozat kiürül, a projektet alkotó minden doboz, és minden csatornapéldányosítása kész, ezt az állapotot *projektindítás kész* állapotnak tekinthetjük.

A projektindítás kész állapot után a dobozok a továbbiakban a csatornákkal önállóan kommunikálnak.

A D-Box projekt működésének leállásának megállapítása nem triviális feladat. Elképzelhető olyan doboz, amely végtelen szimbólumsorozatot ír az output csatornáira. A projekt további dobozai nem feltétlenül fogják ezen adatmennyiséget feldolgozni, van arra lehetőségük, hogy csak annyi adatelemet dolgozzanak fel, amennyire ténylegesen szükségük lehet. Tehát nem mondhatjuk el, hogy a befejezés, leállás feltétele az, hogy a csatornák mindegyikén az *EoC* szimbólum kiírásra és beolvasása megtörtént. Jellemző, hogy az 1-es algráf *memory* output protokollú doboza a *végfeldolgozó* doboz, amely az eredményeket diszkre írja. Azonban általánosságban ez sem igaz: bármelyik doboznak van lehetősége diszkre írni az eredményeket, és *memory* output protokollú dobozból nem csak az 1-es algráfban, egyéb algráfokban is lehet (esetenként akár több) példány is. Az sem kötelező, hogy az ilyen dobozok a bemenő csatornáikról az *EoC* jelekig eljussanak. A *memory* output protokoll miatt annak, hogy ők befejezték a feldolgozást, semmi nyoma nincs.

A projekt működésének befejezését, leállását ezért nem tudjuk triviálisan formális eszközökkel leírni. Általános elvként adhatjuk meg, hogy amennyiben sem a futtató rendszer állapota, sem a futó csatornapéldányok állapota nem változik adott időintervallumban (*timeout*), úgy a rendszert „leálltnak” tekinthetjük.

Szükségesnek tűnik a D-Box projektbe olyan, speciális feladatú csatornák beillesztése, melyeket output célokra a végfeldolgozó dobozok használják, input oldala a futtató rendszerbe vezet. A végfeldolgozó dobozok futásuk végén egyetlen *EoC* szimbólum kiírásával jelezhetik, hogy futásukat befejezettnek nyilvánítják. Amennyiben minden

ilyen csatornán megjelenik az *EoC* szimbólum, a futtató rendszer felismerheti, hogy a projekt befejezte a futását. A D-Box nyelvi leírást ki kell egészíteni annak jelzésével, melyek azok a dobozok, amelyektől ezen jelzések beérkezését várni kell.

Jelenlegi működés szerint amennyiben valamely doboz befejezi a működését (a *Start* kifejezés kiértékelése befejeződik), utolsó előtti lépésében a futtató rendszert értesíti erről, majd a futó kód (.exe) leáll. Ennek alapján a futtató rendszer meg tudja különböztetni a hiba miatt leálló dobozpéldányokat a befejezett futású példányoktól. A programozó, kezelő ki tudja listázni az egyes dobozpéldányok állapotát, és figyelemmel tudja kísérni a végfeldolgozó dobozok állapotát. Amennyiben azok „befejezett, leállt" állapotra váltanak, a maradék, esetleg még futó példányokat leállíthatja (*kill*).

4.7. Összefoglalás

Ebben a fejezetben bemutattuk a D-Box nyelv futtatásához szükséges, általunk fejlesztett futtató rendszer működését. Bemutattuk, milyen információkat tárol a futtató rendszer a futó projektről, hogyan értelmezi és hajtja végre a különböző input és output csatornaindítási és visszakeresési lépéseket.

5. fejezet

Implementáció

Elkészítettük a D-Box nyelvi projekteket fordítani, kódgenerálni, és futtatni képes rendszer egy implementációját. A következőkben leírjuk ezek komponenseit, és működési vázlatát.

5.1. D-Box compiler

Mint ahogy a célkitűzések részben leírtuk, a D-Box nyelv elsősorban a D-Clean nyelvhez készült, alacsonyabb absztrakciós szintű leíró nyelv. Ennek megfelelően az általunk elkészített fordítóprogram alapvetően D-Clean nyelvi forma fordítását végzi. Alkalmas D-Clean nyelven leírt *DistrStart* kifejezésből D-Box nyelvi definíciók generálására, de eleve D-Box nyelven leírt projekt alapján is a fordításra és a kódgenerálásra.

Az általunk fejleszett D-Box fordító lexikális elemzője egy determinisztikus véges automata [40]. A lexikális elemző nem EBNF-ből került generálásra, mivel számunkra a C# programozási nyelv a legtermészetesebb eszköz a fejlesztéshez, és amikor a compiler megírását elkezdtük, még nem volt olyan lexikai elemzőt generáló eszköz, amely az EBNF leírás alapján C# nyelvi kódot generált volna.

A lexikális elemekből felépített struktúra szintaktikai helyességét egy nem tisztán LALR(1)-es szintaktikus elemző ellenőrzi. A hozzá tartozó statikus szemantikai elemző külön menetben működik, csakúgy mint a kódgenerálási menet. Kódoptimalizálási lépéseket pedig nem hajtunk végre, mivel a csatornák kódjában ennek nincs jelentősége, a dobozok kódját Clean nyelven generáljuk, és a kódoptimalizálást a Clean fordító, és lusta kiértékelési rendszer végzi.

A fordításhoz szükséges környezet az alábbi:

- *DCleanComp.exe* fájl a fordítóprogram maga, melynek futásához szükséges a .NET Framework 2.0
- *LocalSett.xml* fájl, az adott gépre specifikus beállítások, melyek elsősorban alkönyvtárak neveit tartalmazzák

- *DCleanConf.xml* fájl, a kódgenerálás során használt beállítások, melyek nem annyira lokális gépbeállításfüggőek, inkább a fordító verziójától függő beállításokat tartalmazzák

A konfigurációs xml fájlok azért kerültek két külön fájlba, hogy a fordító verzióváltásakor, és különböző tesztkörnyezetbe telepítésekor a műveletet egyszerűsítsük. A *LocalSett.XML* fájl cseréje ilyen esetben nem indokolt, gyakran a *DCleanComf.XML* cseréje sem.

A D.1. példában megadunk egy lehetséges *LocalSett.XML* fájl tartalmat. A fájl tartalma gyakorlatilag nem más, mint a fordítás és kódgenerálás során szükséges Clean nyelvi környezet alkönyvtárjainak, és a parancssori fordítók nevei. Hasonlóan tartalmazza a C nyelvi fordító és szerkesztő nevét, szükséges beállításait (pl. include alkönyvtárak nevei). Ezen felül a fejlesztői ICE környezet alkönyvtárjainak és fájljainak neveit.

A D.2. példában a *DCleanConf.XML* fájl egy lehetséges tartalmát adtuk meg. A fájl első felében a Clean rendszer legfontosabb library könyvtárából felhasználható modulok elérhetőségét adjuk meg (a .dcl fájlok szükségesek a típuslevezetéshez), majd a platformspecifikus sablon fájlok neveit. Jelenleg ebben egy bejegyzés található a Windows operációs rendszerű ICE middleware-t használó platform beállításai, de a fordító jelenlegi verziója is képes lenne egyéb platformú megoldások esetén is a kódgenerálásra.

A fordítóprogram paraméterezésének részleteit a D.4. fejezetben tárgyaljuk. A legegyszerűbb paraméterezésű változatban:

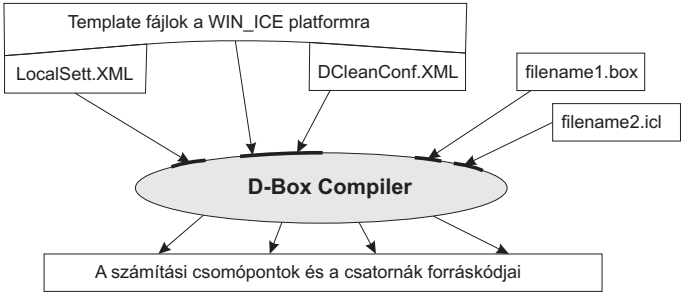
DCleanComp -WDBOX <filename1.box> <filename2.icl>

A *filename1.box* nevű fájl tartalmazza a D-Box nyelvi definíciókat, a *filename2.icl* nevű fájl pedig a boxok belsejében definiált Clean kifejezésekhez tartozó Clean függvények definícióit.

Ennek során a fordító először is beolvassa a két megadott fájl tartalmát. Először a *filename2.icl* fájlban esetleg szereplő *DistrStart* kifejezést eltávolítja. Mivel D-Box fordításkor nem kell típuslevezetést alkalmaznia, így a *filename2.icl* tartalma további elemzésre nem kerül.

A *filename1.box* fájl tartalmazta típusdefiníciók és D-Box definíciók szintaktikai elemzése után következik a statikus szemantikai elemzés. Amennyiben a tartalom hibátlan, úgy a *LocalSett.XML*-beli alkönyvtárnevek felhasználásával a *DCleanConf.XML* fájlban definiáltak szerint a fordító forráskódokat generál.

A generált forráskódok elhelyezéséhez először is nyit egy *./GENERATED* alkönyvtárat, melybe a platformon beállításainál megadott nevek alapján nyitja meg a további alkönyvtárakat. Mivel jelenleg csak a *WIN_ICE* nevű platform esetén vannak a sablon fájlok kidolgozva, így a generált forráskódok a *./GENERATED/WIN_ICE* alkönyvtárba kerülnek.



5.1. ábra. A D-Box Compiler

A fenti alkönyvtárba kerül egy *Makefile* nevű fájl, melyen keresztül ezen platform forráskódjának fordítása elvégezhető. Ezen felül minden egyes alkönyvtárba is kerülnek helyi *Makefile*-ok, mely az adott alkönyvtárban lévő generált forráskódok fordítását végzi el. A külső *Makefile* valójában mást nem is végez, mint a generált alkönyvtárakba megadott sorrendben belép, és az ottani lokális *Makefile* alapján elvégzi a fordítást.

Amennyiben a használt adott nyelvi compiler és linker ezt lehetővé teszi, az általuk kiírt üzeneteket a lokális alkönyvtárbeli *stederr.err* és *stdout.err* fájlokba menti. Amennyiben valamely generált fájl fordítása során hiba lép fel, úgy a hibához tartozó alkönyvtárbeli *stderr.err* fájl fogja tartalmazni a hiba pontos leírását. A hiba javításához ez adhat útmutatót.

5.2. A D-Box sablon alapú kódgenerálása

A kódgenerálás sablon fájlok esetén a következők alapján működik: a D.3. fejezetben megadottak alapján kerül a sablon fájlok tartalma módosításra.

- A D-Box definíciók alapján gyűjtsük ki milyen típusú csatornák vannak használatban a projektben (csak az alap típusnevek érdekesek, [[Int]], [Int] típusú csatornák esetén az *Int* kerül be a listába.
- Gyűjtsük össze a D-Box definíciókat egyetlen listába.
- Generáljunk le minden fájlt, amelyek direktben *Platform*-hoz, *CompNodeShared*-hez, *ChannelShared*-hez, *ServerShared*-hez tartoznak.
- Generáljunk le minden egyes konkrét csatorna-típus esetén a *Channel*-hez tartozó fájlokat, a platform alkönyvtáron belül külön-külön alkönyvtárakat képezve a *Channel_* $<$*TYPENAME*$>$ minta szerint.

- Generáljuk le minden egyes D-Box definícióra a *CompNode* alkönyvtárbeli fájlokat a *Box<BoxID>* alkönyvtárakba, és mindegyikbe generáljuk le a *Start* kifejezést tartalmazó Clean forráskódot is..

5.2. ábra. Sablon fájlok feldolgozása

5.3. D-Box → Clean kódgenerálás

A kódgenerálás a sablon fájlok esetén a definiált makrók feldolgozásában kimerül. A D-Box szemantika helyes implementációja (csatorna-kezelés, a protokollok, a futtató rendszer API hívások) helyessége ezért nagyrészt nem a D-Box fordítón múlik. Ami tényleges (nem, vagy nagyon nehezen sablonizálható) tevékenység, az a számítási csomópontokbeli Clean nyelvi *Start* függvény kódja.

Az 5.3. kódban adtuk meg a generált Clean nyelvi kód általános alakját. A `< ... >` jelek közötti részek kifejtését a későbbiekben ismertetjük.

```
1  Start w
2    #! w                = Middleware_INIT <boxid> w
3      < ide kerülnek az input csatornák indításai >
4    #! inp_channels     = <az input csatornák listája>
5    #! inp_ctypes       = <az input csatornák típusai>
6    #! w                = Channel_StartFINISHED taskINPUT w
7    # input_data_list   = <input protokoll> inp_channels inp_ctypes
8    # result            = < kifejezés alkalmazása >
9       < ide kerülnek az output csatornák indításai >
10   #! output_channels = <az output csatornák listája>
11   #! w                = Channel_StartFINISHED taskOUTPUT w
12   #! w                = <output protokoll> result output_channels w
13   #! w                = Middleware_DONE w
14   = w
```

5.3. példa. Generált Start kifejezés

A *Middleware_INIT* függvény inicializálja a Clean-C interface-t. Ennek során a doboz kapcsolatba lép a *LocalComm* szerverrel, és jelzi, hogy indítási fázisba lépett, ez egyúttal azt is jelzi, hogy hamarosan kezdődnek az input csatornák indításai.

A *Channel_StartFINISHED* függvény hívásával jelzi a doboz a futtató rendszer felé, hogy az input csatornák mindegyikét elindította, vagy felderítette. Ezzel egyúttal jelzi, hogy ezennel áttér az output csatornák indítási fázisára.

Szintén a *Channel_STARTFINISHED* függvény hívásával jelzi, hogy az output csatornák mindegyikének indítását vagy felderítését sikeresen befejezte. Ezek után indul az output protokoll végrehajtása, mely a lusta kiértékelés miatt implicit módon az input csatornák olvasását is indítja.

A *Middleware_DONE* függvény hívásával jelzi, hogy az output protokollt sikeresen végrehajtotta (minden output csatornára kiírta az *EoC* csatorna lezáró jelet). A doboz ezek után már érdemi tevékenységet nem fog végezni, leáll.

A továbbiakban bemutajuk, hogyan működnek az egyes szekciók a fenti kódból.

5.3.1. Az input csatornák indítása

Az input csatornák indításaihoz tartozó Clean kódot attól függően generálja a fordító, hogy milyen típusú csatornákat kell indítani.

- *null* csatorna esetén egyszerűen nem kell generálni semmit,
- *fix* csatorna (pl. ([Int],1002) esetén:

```
#! (inp_chan_0,w) = Channel_FIND 1002 "Int" taskINPUT w
```

- *auto* csatorna (pl. ([Int],auto) esetén:

```
#! (inp_chan_0,w) = Channel_START "Int" taskINPUT 1 w
```

- *autoConnBox* csatorna (pl. (autoConnBox BoxID_1001 ([Int])) esetén:

```
#! (count_0,w) = getThreadCount thisThread "BoxID_1001" threadDIV w
#! (inp_chan_list_0,w) = autoConnBox count_0 ["Int"] w
```

A *Channel_FIND* sablon kód meghívja a futtató rendszert, jelezvén hogy az *1002*-es azonosítójú csatornát keressük, mely „Int” típusú, és a keresést azért végezzük, mert *input* csatornaként kívánjuk használni. Eredményül megkapjuk a csatorna elérhetőségét. A csatorna azonosítóját az *inp_chan_0*-ban tároljuk el. A csatornát egy ICE-ban értelmezett proxy objektum jelöli, melyet a Clean-C interface eltárol, és az azonosítóját (egész szám) adja valójában vissza. Tehát a *inp_chan_0*-ban egy egész szám keletkezik. A továbbiakban ezzel a azonosítóval tudunk a Clean-C interface-n keresztül a csatornára hivatkozni.

A generált kód nem minden esetben *inp_chan_0*, a sorszám a végén a csatorna sorszáma lesz az input csatorna sorozatban. Tehát, ha például már volt 0,1,2 sorszámú csatorna indítás, akkor az új *fix* csatorna azonosítója a *inp_chan_3*-ba kerül.

A *Channel_START* a futtató rendszert utasítja a csatorna indítására, majd az előbbiekhez hasonlóan tárolja az indított csatorna proxy objektumot, és az azonosító sorszámot adja meg.

A *getThreadCount* függvény lekéri az ezen szálban futó (a jelen példa szerint) *BoxID_1001* doboz által indított szálak számát, melyet a *count_0*-ban tárolunk el. A *count* sorszámozása is nő, amennyiben több *autoConnBox* indítást is végre kell hajtanunk.

Az *autoConnBox* függvény adott (jelen példában *count_0*) mennyiségben hajt végre *Channel_FIND* tevékenységet a megadott típusú csatornákkal. Az eredmény nem egyetlen, hanem több csatorna proxy objektumot azonosító szám.

5.3.2. Az input csatornák listájának előállítása

A cél az, hogy az input csatornák indításai során keletkezett csatorna azonosító egész számok egyetlen listába kerüljenek be, mely listán belüli sorrend egyúttal a doboz input csatornáinak sorrendjét is mutatja.

Amennyiben az input csatorna indítások során csak *fix* vagy *auto* csatornák voltak (vagyis csak *inp_chan* értékek keletkeztek), akkor ezeket egyszerűen egy listába kell fűzni. Ha pl. három db ilyen módon indított csatorna volt, akkor az alábbi módon:

```
#! inp_channels = [inp_chan_0, inp_chan_1, inp_chan_2 ]
```

Amennyiben vegyesen vannak alkalmazva a fentiek, úgy előállhat az a helyzet, hogy az indított csatornák azonosítói egyedi *inp_chan*-ek, és *inp_chan_list*-ekbe kerülnek. Ekkor az *input_channels* nem állítható elő az előzőek összefűzésével, mivel ekkor a lista nem annyi elemű lenne, ahány csatorna ténylegesen indításra került. Az alkalmazott megoldás ekkor a *flatten* függvényre épít, mely a listát *kilapítja*, egy dimenziós mélységűvé alakítja, az elemek rekúrzív kifejtése révén:

```
#! inp_channels = flatten [inp_chan_0, inp_chan_list_1, inp_chan_2 ]
```

Amennyiben nem került indításra egyetlen csatorna sem, a fenti lépés kihagyható.

5.3.3. Az input csatornák típusainak listája

A következő lépésben az input csatornák típusait fűzzük össze egyetlen listába. Az *inp_chan_types*-ba pontosan annyi elem kerül be, ahány elemű az előzőekben kialakított *input_channels* lista lett. A típusokat string alakban adjuk meg, például:

```
#! inp_ctypes = ["[Int]", "[[Real]]", "[[Int]]"]
```

Amennyiben nem került indításra egyetlen csatorna sem, a fenti lépés kihagyható.

5.3.4. Az input protokoll alkalmazása

Az input csatornák azonosítói, és típusai ismeretében az input protokollok alkalmazhatók. Mindkét lehetséges input protokoll azonos paraméterezésű, csak nevükben (és természetesen működésükben) különböznek. Vagy a *Join1* vagy a *JoinK* protokoll kerül meghívásra, attól függően melyiket kérték a doboz definícióban:

```
# input_data_list  = Join1 input_channels inp_chan_types
```

vagy

```
# input_data_list  = JoinK input_channels inp_chan_types
```

Amennyiben az input protokoll *memory*, semmilyen sor sem kerül be a generált kódba.

A protokoll függvény eredménye egy *input_data_list*. Ennek előállítása lusta kiértékeléssel történik (hiányzik a strict kiértékelés jele, a !), a csatornák olvasása tehát nem indul el ezen a ponton.

5.3.5. A kifejezés alkalmazása

Az *input_data_list* alapján a dobozba zárt kifejezés már alkalmazható. Az adat-lista a kifejezésnek vagy ebben az alakjában (*joink* esetén) adandó át, vagy elemeire (részlistákra) bontva (join1 esetén). Ez utóbbi esetben annyi rész-listára kell bontani a kifejezést, ahány elem az *input_channels* lista volt:

```
# inp_dat_0 = getDataT0L (input_data_list 0)
```

alakú sorokkal, ahol sorba leválasztjuk az *inp_dat_0*, *inp_dat_1*, stb. rész-listákat, alkalmazva a típusához illeszkedő függvényt. Ugyanis az *input_data_list* egy *Transmissible* típusú elemek, listák listája, tehát a leválasztott al-lista is *transmissible* típusú lenne. Ezt a kifejezésnek paraméterként nem lehet átadni. A fordító ezért generál minden tényleges csatorna-típushoz egy-egy típus-transzformációs függvényt, mellyel az átalakítás elvégezhető. A típusokat sorban *T0*, *T1*, stb. névvel ellátva generál *getDataT0*, *getDataT0L*, *getDataT0LL*, stb. transzformációs függvényeket. Amenyiben a *T0* helyettesíti az *Int* típust, és a csatorna konkrétan *[Int]* típusú lenne, akkor a *T0L* végződésű függvényt kell alkalmazni.

A kifejezésnek megfelelő helyeken átadott paraméter-értékek alapján a kifejezés eredménye (*result*) előállítható (pl. ha a kifejezls `‘first 10‘`):

```
# result = first 10 inp_dat_0
```

Az eredmény előállítása is lusta kiértékelésű, vagyis ez a sor sem indítja még el az input csatornák olvasását.

5.3.6. Az output csatornák indításai

Hasonlóan az input csatornák indításaihoz, az output csatornák indításának szekvenciáját is a fordító generálja, megfelelő sorrendben.

- *null* csatorna esetén nem kell sort generálni
- *auto* csatorna, pl. ([Int],auto) esetén

```
#! (out_chan_0,w) = Channel_START "Int" taskOUTPUT 1 w
```

- *fix* csatorna, pl. ([Int],1002) esetén

```
#! (out_chan_0,w) = Channel_FIND 1002 "Int" taskOUTPUT w
```

- *connBox* esetén

```
#! (out_chan_list_0,w) = connBox "BoxID_1001" ancestorThread taskINPUT w
```

- *startGraph* esetén

```
#! gcount\_0 = expr
#! w = subGraphStartBegin threadDIV w
#! (thread_ids_0,w = startSubGraph 2 threadDIV gcount_0 w
#! w = subGraphStartFinished threadDIV w
```

majd

```
#! (out_chan_list_0,w)
     = getAllChannels "BoxID_1003" thread_ids_0 taskINPUT w
```

A *Channel_START* és *Channel_FIND* hasonló szerepet tölt be, mint az input csatornák indításakor.

A *connBox* függvény lekérdezi az adott szálon futó, adott doboz input szerepkörű csatornáit. Ehhez meg kell várni, amíg az adott doboz jelzi az input csatornák indításának befejezését (*Channel_StartFINISHED*). A *connBox* a lekért csatornák azonosítóit adja vissza lista formában.

A *gcount_0* értéke a *startGraph* kifejezés futási időben kiértékelhető kifejezés eredménye lesz. Ez egy egész szám kell legyen. A *startSubGraph* az adott algráfot indítja el majd ennyi példányban. Ennek során a futtató rendszert kéri fel az algráfhoz tartozó dobozok példányosítására. A *startSubGraph* az indított szálak azonosítóit adja meg. Az eredmény lista annyi elemű lesz, amennyi a *gcount_0* értéke volt.

A *getAllChannels* függvény az adott szálakon futó, adott doboz input csatornáit kérdezi le, és a csatornák azonosítóit adja meg lista alakban.

5.3.7. Az output csatornák típusainak listája

Az output csatornák azonosítóit, hasonlóan az input esethez, listába kel gyűjteni. Teljesen hasonlóan, vagy az elemekből egyszerűen listát fűzünk, vagy szükség esetén alkalmazzuk a *flatten* függvényt.

5.3.8. Az output protokoll alkalmazása

Az output csatornák azonosítói ismeretében az output protokollok alkalmazhatók. Mindhárom lehetséges output protokoll paraméterezése egyezik, ezért jelen formában csak a nevük jelenti a különbséget:

```
#! w = Split1 result output_channels
```

vagy

```
#! w = SplitK result output_channels
```

vagy

```
! w = SplitF result output_channels
```

Amennyiben az output protokoll *memory*, semmilyen sor sem kerül be a generált kódba.

Az output protokoll előtt szerepel a *strict* kiértékelés kérésének jele, vagyis ezen sort a szekvencia ezen a pontján a futtató rendszer elkezdi kiértékelni. Mivel a protokoll függvények paramétere a *result*, így elkezdi kiértékelni a kifejezést is, amelynek azonban az *input_data_list* a paramétere, vagyis elkezdődik az input csatornák olvasása.

A kifejezés paramétereinek kiértékelési jellege (lusta, szigorú) vezérli az input csatornák olvasásának lusta működését. Ha a kifejezés valamely paramétere szigorú kiértékelésű, akkor az adott input csatorna minden egyes eleme ezen a ponton kiolvasásra kerül. Ez nem fejeződhet be, ha az adott input csatorna végtelen adatsorozatot hordoz, mely esetben a kifejezés kiértékelése sem fejeződhet be, így az output protokoll nem küldhet ki egyetlen csatornára sem szimbólumot.

Ha a sorozat véges, ez a szigorú kiértékelési jelleg, akkor is működési korlát, hiszen a doboz további input csatornáinak olvasására addig nem kerül sor, amíg ezen egyelen csatorna olvasása be nem fejeződik. Ezt az input csatornát tápláló doboznak, amely esetleg a többi input csatornát is táplálná, folyamatosan munkát ad, miközben egyéb output csatornáit elhanyagolhatja. Ezért a kifejezés paraméterezésében csak indokolt esetben használunk szigorú kiértékelést.

Többszálú protokollimplementáció esetén az előzőekben ismertetett elvek érvényüket vesztik. A csatornák olvasása történhet párhuzamosan is. A D-Box nyelv szemantikája nem kényszerít sem egyik, sem másik típusú implementációra.

5.4. A D-Box nyelv implementációjának általánosítása

A fentiekben bemutattuk, hogy a D-Box koordinációs nyelv hogyan kerül implementálásra *Clean* nyelv esetén. Az implementálás részleteit a külső sablon fájlok rejtik, melyekben szereplő sablonokat a D-Box kódgenerálás során makródefiníciók segítségével a fordító végzi.

A D-Box nyelvi szemantika implementációja ennek következtében a sablonokba kerül. A sablonok elkészíthetők más funkcionális nyelvekre, és más köztes réteg, más futtató rendszer esetén is. A sablonok írásához makrókat kell használni, melyek a D.3. függelékben kerülnek bemutatásra. Amennyiben az eltérő környezetbeli implementálás során a meglévő makrók nem elegendőek, úgy a D-Box fordítót új makrókkal lehet kiegészíteni.

A D-Box típusrendszer két alappillére a *D-Box nyelvi alaptípus*, és a *helyreállítható* típusok halmaza. Az alaptípusokból rekord- és listaképző konstruktorok segítségével áll elő a *szállítható* típusok halmaza. Ezen típusokra van jelenleg a D-Box makrónyelv felkészítve. Amennyiben más típusok kezelésére is szükség van, úgy a D-Box fordító makrókészlete bővítésre kell kerüljön. A bővített makrók alapján a sablonok elkészíthetők, és a nyelv máris képes újabb típusok kezelésére is.

A dolgozat szövegezésében felsorolt nyelvi alaptípusok halmazának meghatározását egyrészt a *Clean* nyelvi típusok, és az ICE köztes réteghez tartozó SLICE interface leíró nyelvi típusok befolyásolták. A rekord és a lista típuskonstruktorok azonban a D-Box nyelvhez tartozóan értelmezettek, vagyis újabb nyelvi alaptípus bevonása esetén azokra automatikusan vonatkoznak.

A nyelvi alaptípusokhoz típusmegfeleltetéseket kell alkalmazni. A D-Box nyelv típusnevei nem feltétlenül egyeznek meg az alkalmazott funkcionális nyelv típusneveivel. Valójában a jelenlegi Clean + ICE implementáció során a D-Box nyelvi típusneveket még a C++ nyelvi típusoknak is meg kell feleltetni, mivel a kommunikáció során nem direkt módon a Clean-ből hívjuk meg az ICE köztes réteg szolgáltatásait, hanem C++ interface került illesztésre a Clean és az ICE közé. A D-Box nyelv beépített típusmegfeleltetési táblázatot használ a D-Box típusnevek, a Clean és a C++ típusneveket illetően. Ezen táblázat egyéb nyelvek felé történő kiterjesztés esetén bővítésre szorulnak.

Az input protokollok mindegyike egyező paraméterezésű, meg akkor is, ha ez kicsit értelmet zavaró lehet első olvasáskor. A *join1* protokoll első paramétere n darab csatorna, második paramétere n darab típus neve. A *joink* első paramétere szintén n darab csatorna, második paramétere szintén n darab típus neve, holott a *joink* esetén minden típusnév egyforma kell legyen. Ennek a látszólagos bonyolításnak az oka, hogy újabb input protokoll esetén a rendszer egyszerűen bővíthető legyen: mindössze az újabb protokollt is erre a paraméterezési mintára kell felkészíteni. Amit a D-Box fordítónak valójában ismernie kell a statikus szemantikai helyesség elemzéséhez, az a

protokoll eredményének értelmezése. A *join1* eredménye egy n különböző érték (melyeknek akár a típusúaik is különbözőek lehetnek), a *joink* eredménye pedig egyetlen lista lesz. Ezt kell az input wrappernek átadni, aki a helyreállítható értékek listájával összefuttatva elő tudja állítani a kifejezés argumentumait.

Hasonlóan, az output protokollok paraméterezése is megegyezik. Mind a *split1*, mint a *splitk*, mind a *splitf* első paramétere adatok egy listája, második paramétere csatornák egy listája. A *splitk* esetén az első paraméter valójában egyetlen érték kellene legyen, hiszen a *splitk* ezen értéket allistákra bontja, és úgy dolgozza fel. Új output protokoll esetén maga a D-Box kódgenerálás ide vonatkozó része nem szenved jelentős módosítást: amennyiben az új protokoll is ugyanilyen paraméterezésű lesz. A statikus szemantikai ellenőrzést természetesen ki kell terjeszteni az új protokoll értelmezése alapján is.

Ugyanakkor a protokollok működésének illeszkednie kell a D-Box specifikációra, de a D-Box fordító és kódgeneráló az implementáció részleteit nem tartalmazza. A protokollok implementációja sablon fájlokban van, melyek függhetnek az alkalmazott funkcionális nyelv szintaktikájától és szemantikájától. A *Clean* nyelvi implementáció pl. megvalósítható lett volna az *ObjectIO* támogatásával is. Jelen megvalósítása e helyett a lusta kiértékelésre épül, és a jelenlegi input és output protokollok működését sikerült is ezen kiértékelési rendszerre alapozni. (Bár az output protokoll implementálásához szükséges volt még a futtató rendszer támogatása is a szabad csatorna kiválasztásához).

5.5. Futtató rendszer implementációja

A futtató rendszert teljes egészében C#-ban fejlesztettük ki. Ez egy OOP alapokra épülő, a Microsoft.NET programozási platformon használható programozási nyelv. A generált kód virtuális futtató rendszer meglétét feltételezi az őt futtató számítógépen, melyet *.NET Framework*-nek nevezünk. A framework minden windows operációs rendszerhez rendelkezése áll, de a linux-hoz is létezik implementációja [6]. Bár a linux implementáció nem teljes (főként a WinForm támogatás hiányos), de a futtató rendszer komponensei konzolos felületű alkalmazások, melyek futtathatóak akár a Linux Mono implementáción is.

A futtató rendszer komponensei egymással a távoli metódushívás módszerrel kommunikálnak. Erre a framework tartalmaz saját megoldást, melyet Microsoft RPC modellnek nevezhetünk. Ennek fejletteb módszerei fedezhetőek fel a Windows Communication Foundation-ban, melyet azonban csak a 3.0 verziójú framework-ben jelentett meg 2006 júniusában a Microsoft hivatalosan (a projekt korábbi neve Indigo volt, és a fejlesztés különböző fázisaiban lehetett vele korábban is találkozni).

Mivel a futtató rendszer fejlesztésekor még nem állt rendelkezésre a WCF, és nem

kívántunk a Microsoft RPC-t használni (mellyel teljesen kizártuk volna más platformok és más programozási nyelvek használatát a későbbiekben), így más megoldást kellett keresnünk a komponensek közötti kommunikáció megoldására.

A CORBA elméletileg pontosan lefedte volna a felmerülő igényeket, de a Microsoft nem adott implementációt a CORBA protokollok használatára a .NET Framework egyik verziójában sem. Ennek vélhető oka, hogy nem tagja a OMG-nek. Emiatt pontosan a C#-t zártam volna ki a fejlesztésből.

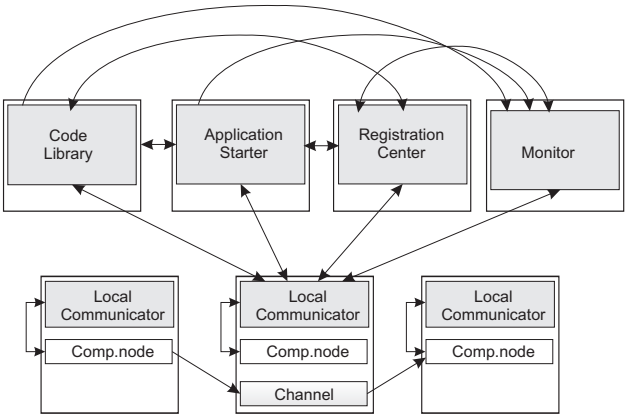
A választás az ekkoriban gyerekbetegségein már túllépő, ígéretes jellemzőkkel bíró, GPL licenc alatt hozzáférhető, jól dokumentált *ICE* [38][39] köztes rétegre esett. Ez a CORBA-hoz igen hasonló, jó teljesítménymutatókkal rendelkező megoldás rendelkezett C# (pontosabban .NET Framework) támogatással, implementálva volt windows és linux platformokon túl számos mobil eszközre is. Támogatással bír C++, Java, .NET, Python, PHP, Ruby programozási nyelvekre. Saját interface leíró nyelve van, melyet SLICE-nak neveznek, és windows-os rendszerekben futtatási környezet telepítése nagyon egyszerűen történik, mindössze három DLL fájlt kell bemásolni a C:/Windows/System32 alkönyvtárba.

Az ICE-nak sajnos nincs Clean nyelvi interfésze, de erre igazából nem is volt szükség. A Clean-hez ugyanis külső C nyelvi object állományok szerkeszthetőek, amely C nyelvi függvények viszont a Clean-ból hívhatóak. Az ICE-nek pedig volt C nyelvi interfésze, így végül is Clean-ből a szükséges funkciók meghívhatóak.

Az ICE kiegészítője az ICE-E környezet, melyben számos utility segít egy komplex megoldás kiépítésében. Sajnálatos módon az ICE-E futás közbeni támogatása elsősorban a szolgáltatás orientált rendszerek felé irányul. Szerver (szolgáltatást nyújtó) alkalmazások telepítését, konfigurálását, és felderítését támogatja, bizonyos fokú verziókövetéssel. Ennek központi eleme az *IceGrid*, mely grafikus felülettel rendelkező konfigurációs szoftver, amelynek segítségével a *grid* elemeken futó szolgáltatások és komponensek konfigurálhatóak.

Ugyanakkor egy fejlesztő-futtató jellegű projektben, mint amilyen a D-Box jellegű projektek, nem könnyen alkalmazható ez a módszer. Nincs szükség verziókövetésre, sőt, káros lenne. A D-Box projektek egyszeri futásúak, a befejezés után minden részei (dobozok, csatornák) le kell álljanak. A komponensek nem előre meghatározott gépekre kerülnek, hanem indításuk után dinamikusan kerül kiválasztásra melyik gépen kerülnek futtatásra. Ennek oka, hogy a dinamikus algráf indítások miatt nem készülhet el statikus futtatási terv. Ezen kívül is szükség van bizonyos speciális D-Box műveletek támogatására, mint az egyedi csatornaazonosítók kiosztása, algráfok indítása futás közben, stb.

A D-Box nyelvi programok elindítására, és futás közbeni műveletei támogatására ezért egy speciális futtató rendszer, az (*Application Environment – AppEnv*) került kifejlesztésre.



5.4. ábra. Futtató rendszer sematikus rajza

Egy D-Box projektek életciklusa, fontosabb tevékenységei az alábbiakkal jellemezhető:

- A projekt fordítása után (amelynek során van lehetőség különböző platformú futtatható állományok előállítására is) a kapott bináris állományok a megfelelő fordító és szerkesztő programok használata mellett előállnak.
- A bináris állományokat a fejlesztő feltölti egy kódtárba (CodeLib)
- A projekt futtatását a fejlesztő vagy tesztelő kéri.
- A futtatási kérést egy ütemező (AppStarter) kapja meg, amely lekéri, hogy a projekt indításához (*1-es algráf elemei*) milyen platformra vannak generált bináris kódok.
- Az ütemező lekéri az on-line számítási node-ok platform jellemzőit, összehasonlítja a szükséges tevékenységek listájával, majd futtatási tervet készít (melyik bináris fájlt melyik node-on kell futtatni).
- Amennyiben a futtatási terv elkészíthető, úgy kezdeményezi a kiválasztott node-okon a megadott projekt elemek letöltését és indítását.
- Az indított D-Box csomópontok kapcsolatba lépnek a futtató rendszerrel, közlik a csatorna igényeiket, melyek vagy csatornaindítási kérelmek, vagy csatornakeresési kérelmek.

- A futtató rendszer névszolgáltatója (RegCenter) tárolja a futó projektelemek azonosítóit (projektazonosító, szálazonosító, doboz- vagy csatornaazonosító). A keresési kérelmek, amennyiben már regisztrált elemre mutatnak, úgy egy lokációt azonosító válasszal kielégíthetőek. A csatornaindítási kérelmek azonban újabb kódletöltéseket és indítási utasításokat generálnak. A frissen indított csatornák is a névszolgáltatóba regisztrálják be magukat.

- A csomópontokat alkotó számítógépeken a futtató rendszer egy speciális szolgáltatása (LocalComm) fut, amelyhez a „töltsd le" és „futtasd" alaputasításokat ki lehet adni. Ezen szolgáltatás egyéb szerepeket is betölt, de legfontosabb tevékenysége a projektelemek adott gépen történő indításának támogatása.

A fejlesztett futtató rendszer részét képezi még egy Monitor, amely on-line eseménykövetést tesz lehetővé, valamint log-ot is ír. Az utasításokat a fejlesztő pedig egy AppEnv parancssori (*command line*) segédprogram segítségével képes kiadni.

Jegyezzük meg, hogy a futtató rendszer kifejlesztése a D-Box projektek teszteléséhez szükséges, de nem volt elsődleges feladata a dolgozatnak. Ezért a tervezés során sok olyan pont is felbukkan, amely nyitottá, továbbfejleszthetővé teszi ezt a rendszert, de ezek a fejlesztések nem kerültek implementálásra. Ilyen például a több-platform esetén kidolgozottság. Az *AppEnv* szerverek API függvényreferenciáját a C. fejezetben adjuk meg.

5.5.1. A futtató rendszer jellemzői

A futtató rendszer komponensei a *LocalComm*-t kivéve egy példányban léteznek egyetlen alhálózat esetén. A komponensek a futásuk során begyűlt információkat a memóriában tárolják, ez alól kivételt csak a *CodeLib* képez, ami a feltöltött kódokat az őt futtató számítógép fájlrendszerében tárolja el. Ez utóbbit tekinthetjük a rendszer erőforrásainak.

Az alábbiakban tárgyaljuk a korábbiakban leírt futtató rendszer jellemzőit[43]

- *Hozzáférhetőségi átlátszóság*: a rendszer az ICE middleware működésén keresztül ezt teljesíti, a kommunikáció során a SLICE interface leíró nyelv által ismert típusokat a middleware átlátszóan kezeli

- *Elhelyezkedési átlátszóság*: a rendszerbeli névszolgáltató (*RegCenter*) 5 másodpercenként az alhálózatra broadcast üzeneteket küld ki, melyek felismerése révén a névszolgáltató helye felderíthető. A további futtató rendszer komponensek is innen azonosítják a névszolgáltatótó helyét, majd saját magukat regisztrálják. Emiatt a futtató rendszerbeli komponensek helye a névszolgáltatótól lekérhető. A komponensek fizikai elhelyezkedése azonban nincs elrejtve az őket használó futó

programok elől, mivel a névszolgáltatótól lekért elérhetőségek IP címet és portot tartalmaznak.

- *Áthelyezhetőségi átlátszóság*: A szolgáltatásokat használó programok elsősorban a névszolgáltatótól érkező broadcast üzenetre várakoznak. Ezek után a további szolgáltatások címeit a névszolgáltatótól kérik le, ezért maga a névszolgáltató, és a szolgáltatások korábbi helyeit nem ismerhetik, nem is fontosak számukra. Egy konkrét szolgáltatás címének ismeretében a szolgáltatás igénybe vehető. Ha később újra szükséges számára az adott szolgáltatás, akkor a korábbi cím birtokában próbálkozhat annak folytatólagos használatával. Sikertelenség esetén az eljárás hasonló: meg kell várni a következő broadcast üzenetet, majd a névszolgáltatótól a szolgáltatás új címét lekérni. Ez esetben tudomásul kell venni, hogy az új szolgáltatás a korábbi együttműködés során átadott információkat nem tartalmazza.

- *Mozgathatósági átlátszóság*: a futtató rendszer aktív elemeinek áthelyezése nem megoldható, mivel nem képesek saját állapotukat lementeni és visszatölteni. Az inaktív elemek leállítása és az új helyen (manuális) elindítása azonban általában megoldható az alábbiak szerint:

  - *RegCenter* inaktív, ha nincs folyamatban futó projekt. Futó projekt esetén a *regcenter* tárolja a már indított dobozpéldányok és csatornapéldányok azonosítóit és helyét, így leállása a projekt működésképtelenné válását okozhatja. Ha inaktív, akkor új gépen történő indítását követően broadcast üzenetekkel tájékozatja a komponenseket az új elérhetőségéről, akik az új *RegCenter* példányba újra beregisztrálják magukat.
  - *AppStarter* inaktív, ha nincs folyamatban projekt vagy algráf indítása. A projekt a futása során dinamikus módon algráfokat indíthat, melyekhez szintén szükséges az *AppStarter* szolgáltatás. Ha egyik sincs folyamatban, az *AppStarter* leállítható, és új gépen elindítható.
  - *CodeLib* inaktív, ha nincs projekt vagy algráf indítás folyamatban. Inaktív állapotában leállítható, és új gépen elindítható. A korábbi gép fájlrendszerében tárolt kódokat is át kell másolni az új helyre, mely esetben a fájlrendszer felderítése után újra képes kiszolgálni a letöltési kéréseket.
  - *Monitor* inaktív, ha nincs folyamatban logolási tevékenység, vagyis ha nincs projekt futtatva. A beérkező üzeneteket fájlba írja, így újraindítása után másik log fájlt készít. A korábbi eseményeket az előző log fájlból, az új események az új log fájlból kiolvashatók.
  - *LocalComm* inaktív, ha az adott gépen nem fut egyetlen doboz- vagy csatornapéldány sem. Leállása, és másik gépen elindítása ugyanakkor nem minősül

áthelyezésnek, inkább a régi gép kiesése után az új gép hálózatba kapcsolásaként értelmezhető.

- *Többszörözhetőségi átlátszóság*: a futtató rendszerbeli elemek az alábbiak szerint lehetnek többszörözve:
  - *RegCenter* egyetlen példányban lehet egy alhálózat esetén.
  - *AppStarter* több példányban is lehet egy alhálózat esetén, a vele történő műveletvégzés mindig egyetlen függvényhívást jelent (algráf vagy projekt indítása). Nem jelent gondot, hogy akár ugyanazon projekt különböző algráf indításait más-más *AppStarter* végzi. A különböző *AppStarter*ek címei különbözőek. Műveletvégzés előtt le lehet kérni egy konkrét *AppStarter* címét, és felkérni a művelet elvégzésére.
  - *CodeLib* több példányban is lehet egy alhálózat esetén. Ilyenkor jellemzően más-más platformra vonatkozva tárolják a szükséges kódokat. Az *AppStarter* választja ki egy adott doboz- vagy csatornapéldány indítása során melyik *CodeLib* lesz igénybe véve. A *CodeLib* szolgáltatások címei különbözőek.
  - *Monitor* egy példányban létezhet.
  - *LocalComm* jellemzően több példányban léteznek, címeik különbözőek. Ezen felül különbözhetnek abban is, milyen platform tartozik hozzájuk. Ugyanazon a számítógépen azonban csak egy *LocalComm* futhat egy időben.

- *Egyidejűségi átlátszóság*: a futtató rendszer bizonyos szolgáltatásai nem érhetők el egy időben, de ekkor a rendszer saját maga gondoskodik a zárolásokról, és az ütemezésről. Tranzakciók nincsenek értelmezve a rendszerben.

- *Meghibásodási átlátszóság*: a futtató rendszerek meghibásodási védelme nincs kidolgozva. Nincsenek automatikus újradindítási mechanizmusok. Ha valamely szolgáltatás leáll, akkor azt csak manuálisan lehet újraindítani. Ha aktív állapotban állt le, akkor a szolgáltatást használók valószínűleg nem képesek hibátlanul befejezni a saját működésüket. A leállt szolgáltatásokat a lelassultaktól meg lehet különböztetni azon okból, hogy a leállt szolgáltatások a korábbi címükön (ip cím és port) nem érhetők el, vagyis az operációs rendszer hibát jelez a velük való kommunikációs próbálkozás esetén.

- *Állandósági átlátszóság*: a futtató rendszer részei folyamatosan a memóriában vannak, így ezen jellemző nem vizsgálható.

- *Nyitottság*: a futtató rendszer szolgáltatásainak szintaktikáját a SLICE nyelvi definíciók adják meg, ezért ezen szempont szerint a rendszer nyitott. A szemantikai leírását ezen disszertáció tartalmazza.

- *Átméretezhetőség*: a gépek rendszerbe bekapcsolása új *LocalComm* szolgáltatások indításával folyamatosan bővíthető. A frissen indított gépek passzívan várakoznak a broadcast üzenetekre, majd saját magukat a *RegCenter*be regisztrálják. Több alhálózat összekapcsolásához az adott alhálózatokban üzemelő *RegCenter*ek összekapcsolása lenne szükséges, mely egyelőre nem került kidolgozásra.

5.5.2. A D-Box nyelv elosztott jellemzői

- *Hozzáférhetőségi átlátszóság*: a D-Box nyelv a 2.3. definícióban megadott típusú adatokat képes a dobozok között szállítani. Ezen típusokra a hozzáférhetőségi átlátszóság teljes. Egyéb típusok esetén nincs értelme ezen kérdést vizsgálni, mivel nem képes a szállításra.

- *Elhelyezkedési átlátszóság*: a D-Box nyelven a dobozokra azonosítójukkal, a csatornákra vagy azonosítójukkal, vagy implicit módon a doboz, azon belül a csatorna sorszámával hivatkozik. A D-Box nyelven a példányok azonosítására más mód nem létezik. Ezen azonosítási módszer mellett a fizikai hely ismerete nem szükséges: azt majd a futtató rendszer fogja meghatározni. Az elhelyezkedési átlátszóság tehát teljes.

- *Áthelyezhetőségi átlátszóság*: Sem a doboz, sem a csatornapéldányok a projekt futása során nem helyezhetők át. Ezt szempontot a rendszer nem teljesíti.

- *Mozgathatósági átlátszóság*: hasonlóan az áthelyezhetőségi átlátszóság esetén említettekhez: ezt a szempontot a rendszer nem teljesíti.

- *Többszörözhetőségi átlátszóság*: a dobozpéldányok futás közbeni azonosítójában a szálazonosító is szerepel. Emiatt, bár egy dobozból több példány is készülhet, ezen példányok a szálazonosítóban különböznek. Emiatt elmondható, hogy minden futás közbeni példány egyedinek tekinthető, és többszörözésről nem beszélhetünk. A csatornák esetén sem létezhet ugyanazon csatorna több példányban. Ezen szempontot a rendszer tehát nem teljesíti.

- *Egyidejűségi átlátszóság*: mivel nincs többszörözöttség a rendszerben, a megoszthatósági szempont nem vizsgálható.

- *Meghibásodási átlátszóság*: valamely csatorna vagy doboz példány meghibásodása a rendszerben a teljes rendszer meghibásodásához vezet. Ez alól kivételt képeznek azok az esetek, amikor olyan doboz vagy csatorna hibásodik meg, amely a további futás alatt már nem végezne kommunikációt. Ezen esetekben a meghibásodás észrevétlen, nem okoz semmilyen hatást a rendszeren.

- *Állandósági átlátszóság*: a doboz- és csatornapéldányok folyamatosan a memóriában vannak, így ez a jellemző nem vizsgálható.

- *Nyitottság*: a D-Box nyelv kódgenerálási képessége a külső sablon források módosításával más platform és más köztes réteg esetére kidolgozható. Az XML konfigurációs fájlba az új platform és köztes réteg esetén a módosítások átvezethetőek, a fordító fog tudni dolgozni ezekkel a séma fájlokkal. Ez alól kivételt képez a *Start* függvény generálásának menete, amely azonban nem tartalaz sem platform, sem köztes réteg függő kódot. Módosítani kell azonban a fordítóprogramot abban az esetben, ha a külső sémafájlok tartalmi kitöltése új makrók bevezetését igényli. A fordító és kódgeneráló rész, a makrók működése ezen disszertáció témája. A D-Box nyelv szintaktikai és szemantikai ismertetése a publikációkban már részben lefedésre került, valamint ezen disszertációban is tárgyalásra került.

- *Átméretezhetőség*: a D-Box rendszerbeli egyedek (dobozok, csatornák) száma nem korlátos, tetszőlegesen nagy projekt leírható segítségével. A jelenlegi platform és köztes réteg működésének alapjául a TCP/IP protokoll szolgál, mely lehetővé teszi sok gépes környezetben is az egyértelmű azonosítást.

5.6. Összefoglalás

Ebben a fejezetben bemutattuk, hogyan kerültek implementálásra a D-Box nyelv és a futtató rendszer szolgáltatásai. A D-Box nyelvi protokollok implementálása sablon fájlokban a DVD mellékleten megtalálhatóak. A Clean → C → ICE interface sablon fájlok szintén a lemezmellékleten találhatók meg, csakúgy mint a csatorna sablon fájlok. Az integrált D-Clean, D-Box fordító konfigurációs XML fájlok tartalma és a parancssori paraméterezése is ismertetésre került. A generált Clean nyelvű Start kifejezés előállítása szintén bemutatásra került. Az elosztott futtatás és működés támogatására készített futtató rendszer részei, jellemzése ismertetésre kerültek. A futtató rendszer API bemutatása helyhiány miatt nem ebben a fejezetben, hanem a C. függelékben kapott helyet.

6. fejezet

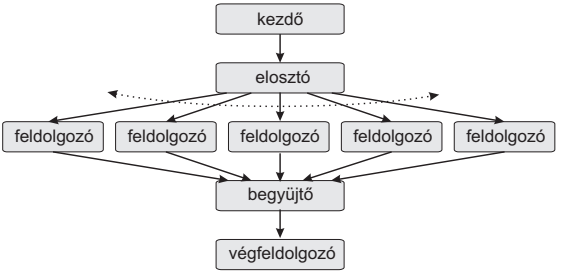
Tesztelés

A D-Box nyelvű program a futtató rendszeren végzett teszteléséről az 1. és 9. publikációkban is lehet olvasni. További tesztelésekhez egy egyszerű programozási feladatot választottunk: a visszalépéses keresés algoritmusával megoldható N *királynő* problémájának egy implementációját. A visszalépéses keresés általános feladata: válasszunk ki N sorozatból 1-1 értéket oly módon, hogy a kiválasztott értékek ne kerüljenek kölcsönösen kizáró helyzetbe. Az N királynő esetében az N sorozatot az $[1 \dots N]$ értékek képviselik, az i. sorozatbeli j érték választása azt jelenti, hogy az i. sorban a j. oszlopba helyeztük el a királynőt. Egy i, j választás ezért y, x koordinátaként is felfoghatunk. Egy y_j, x_j választás nem kizáró helyzetű egy y_k, x_k választással ($j \neq k$), ha a két királynő nem kerül ütési helyzetbe a sakktáblán: vagyis $y_j \neq y_k$ (nem egy sorban vannak), $x_j \neq x_k$ (nem egy oszlopban vannak), valamint $|x_j - x_k| \neq |y_j - y_k|$ (nincsenek átlós helyzetben).

A konkrét megoldás során a ténylegesen meghatározásra kerül adott N ismeretében a lehetséges megoldások száma. Az algoritmus exponenciális időigényű, ennek következtében a méréseket szűk paraméterváltozási környezetben tudtuk tesztelni: $N \in [10{,}15]$ esetekben. Egy megoldás kész, ha az $N \times N$-es táblán az N királynő sikeresen elhelyezésre kerül.

Az algoritmust módosítottuk, hogy elosztott körülmények között is működni tudjon. Ennek során generálunk egy *megoldáskezdeményt*, amely $K < N$ királynőt helyez el a táblán, majd ezen K elemű megoldáskezdeményeket a maradék $N - K$ királynő elhelyezésével folytatjuk. A megoldáskezdeményeket egy generáló doboz folyamatosan állítja elő, majd egy elosztó doboznak ad át. Az elosztó doboz M feldolgozó dobozzal áll kapcsolatban, a megoldáskezdeményeket mint befejezendő feladatokat folyamatosan továbbítja nekik. A *SplitF* protokoll biztosítja azt, hogy a feldolgozó dobozok egyenletes terheléssel kapják meg a számítási feladatokat. A megoldó dobozok mindegyike előállítja, hogy ő hány darab megoldást talált, majd ezt az egész szám értéket továbbítja egy begyűjtő doboz felé. A begyűjtő doboz összesíti az eredményeket, és egy végfeldolgozó doboz felé adja tovább. A végfeldolgozó doboz a kapott értéket egy fájlba írja ki (6.1.

ábra).



6.1. ábra. D-Box működés

A vezérek (királynő) helyzetét a sakktáblán egy rekord írja le, melyben szerepel a vezér x,y koorintája (sor,oszlop), valamint hogy hanyadik elemet választottuk ki az adott sorozatból (s). Az N királynő probléma egyik kulcs függvénye a „nincs_utesben". Ezen függvény paramétereként a következő elhelyezendő vezért (r) veti össze az eddig elhelyezett vezérekkel, melyet a második paraméter hordoz (6.2. példa).

A következő elem választását egy adott részsorozatból a „kovJoLepes" függvény végzi. Paramétereként kapja az eddig elhelyezett vezérek pozícióját hordozó listát, és második paramétereként a következő (esetleg jó) választás sorszámát. Magát a sorozatot, amelyből választania kell, nem kapja meg paraméterként: az esetünkben mindig az $[1..N]$ sorozat. A függvény egy *Vezer* rekorddal tér vissza, melynek x,y koordinátája 0 értékű, ha a választás nem sikerült, ellenkező esetben egy jó választást leíró koordinátával tér vissza (6.3. példa).

Clean nyelvi függvény

```
1  :: Vezer = {x::Int, y::Int, s::Int}
2
3  nincs_utesben :: Vezer [Vezer] -> Bool
4  nincs_utesben _  []    = True
5  nincs_utesben r [h:tl]
6    | (r.x==h.x) = False
7    # dx = abs (r.x-h.x)
8    # dy = abs (r.y-h.y)
9    | (dx==dy)   = False
10   = nincs_utesben r tl
```

6.2. példa. „Nincs ütésben" függvény

Clean nyelvi függvény

```
1  kovJoLepes::[Vezer] Int -> Vezer
2  kovJoLepes [r:tl] i
3    | i>N              = {x=0,y=0,s=0}
4    # n                = {x=r.x, y=r.y, s=i}
5    # ok               = nincs_utesben n [r:tl]
6    | not ok           = kovJoLepes [r:tl] (i+1)
7    = n
```

6.3. példa. „kovJoLepes” függvény

Magát a visszalépéses keresést a „backtrack” függvény végzi. Paramétereként kap

- egy listát a korábban már jól elhelyezett vezérekről,
- a minimális és maximális mélységet,
- az első paraméterben adott lista elemszámát,
- a következő (esetleg jó) választás sorszámát.

Az első paraméterben egy $l <= N$ elemszámú listát adunk át. Mivel az elosztott feldolgozás során valamely k lépésig kész megoldáskezdeményről indul a backtrack függvény, a $min = k$ korlátozza a visszalépés mélységét. A $max = N$ esetben pedig előírjuk a függvénynek, hogy igyekezzen elérni az N elemszámot, ekkor lesz sikeres a megoldás keresése (4. sor). A $min = 1$, $max = N$ esetben beszélhetünk a hagyományos működésről[1], mely esetben az algoritmus maga állítja elő a teljes megoldássorozatot. Amennyiben például $N = 10$, $min = 1$, $max = 3$[2], úgy egy 10×10-es sakktáblán 3 vezér elhelyzésekor sikeres működésről beszélhetünk. Ez esetben a függvény generál egy megoldást, melyen az első három sorban egy-egy vezért helyez el. Amenyiben például $N = 10$, $min = 3$, $max = 10$[3], úgy a függvény megpróbál megoldást keresni oly módon, hogy 3 mélység alá sosem lép. Ha a keresés közben 3 mélység alá kellene lépnie úgy sikertelen a megoldás keresése (6.4. példa).

A *backtrack* függvény adott megoldáskezdemény alapján vagy talál megoldást (4. sor), vagy ha nincs megoldás, akkor üres listával tér vissza (7. sor). A megoldás keresése során a *kovJoLepes* függvény alapján próbálja elhelyezni a vezért az adott sorban. Ha nem sikerül ebben a sorban a vezér elhelyezése (6. sor ellenőrzés), akkor visszalépünk egy mélységet (8. sor). Ha sikeres volt az elhelyezés, akkor előrelépünk (9. sor).

Az összes megoldás generálását a „backtrack_all” függvény kényszeríti ki (6.5. példa). Paramétereként kapja a korábban már bemutatott *min*, *max* értékeket, a megoldáskezdemény hosszát, magát a következő vezér következő (esetleg jó) helyét a soron belül,

[1] Ez esetben a megoldáskezdemény az üres lista.
[2] A megoldás-kezdemény is üres lista.
[3] Egy három elemű megoldáskezdeményt adunk át.

Clean nyelvi függvény

```
1 backtrack::[Vezer] Int Int Int Int -> [Vezer]
2 backtrack [] _ _ _ _   = []
3 backtrack [r:tl] min max length i
4   | length>= max  = [r:tl]
5   # next          = kovJoLepes [r:tl] i
6   | next.s == 0
7      | length==min= []
8      | otherwise  = backtrack tl min max (length-1) (r.s+1)
9   | otherwise     = backtrack [next:r:tl] min max (length+1) 1
```

6.4. példa. „backtrack” függvény

illetve magát a megoldáskezdeményt. Első lépésként meghívja a *backtrack* függvényt, hogy készítsen megoldást. Amennyiben üres listával tér vissza (4. sor), a *backtrack_all* függvény is befejezi a működését. Amennyiben van megoldás, úgy azt a 8. sorban a megoldáslista elejére fűzi, majd folytatja rekurzívan a további megoldások keresését (7. sor).

Clean nyelvi függvény

```
1 backtrack_all::Int Int Int Int [Vezer] -> [[Vezer]]
2 backtrack_all min max length step basic
3  # sol          = backtrack basic min max length step
4  | isEmpty sol  = []
5  # r            = hd sol
6  # tail         = tl sol
7  # rems         = backtrack_all min max (max-1) (r.s+1) tail
8  = [sol:rems]
```

6.5. példa. „backtrack_all” függvény

Az elosztott működéshez szükséges, a 6.1. ábrán látható dobozok belsejébe kerülő függvényeket a 6.6. példában mutatjuk be:

- a *start_gen* függvény kerül a kezdő dobozba. A *cut* paraméter értéke határozza meg a generált megoldás-kezdemények elemszámát,
- az *divider* függvény kerül az elosztó dobozba,
- a *worker* függvény kerül a feldolgozó dobozba,
- a *merger* függvény a begyűjtő dobozba
- a *saver* függvény pedig a végfeldolgozó dobozba.

Clean nyelvi függvény

```
start_gen::[[Vezer]]
start_gen = solutionsB
  where
     solutionsB
       = flatten (map (backtrack_all 1 cut 1 1) solutionsF)
     solutionsF
       = [ [{x=xx, y=1, s=xx}] \\ xx <- [1..tabla] ]

divider::[[Vezer]]->[[Vezer]]
divider ls = ls

saver::Int *World -> *World
saver all_solutions w
  # (f,w) = openFile w
  # f     = WriteToFile all_solutions f
  = closeFile f w

merger::[Int]->Int
merger [] = 0
merger [a:tl] = a + (merger tl)

worker::[[Vezer]]->Int
worker basic_solution = length (flatten solutions)
 where
   solutions
     = map (backtrack_all cut tabla cut 1) basic_solution
```

6.6. példa. doboz-függvények

A dobozok D-Box nyelvi kódját 4 feldolgozó doboz esetén a G. függelék tartalmazza.

6.1. Az elosztott működés tervezése

Az elosztott működéshez a *start_gen* függvény készíti el a megoldáskezdeményeket, melyek alapján a feldolgozó dobozok a megoldásokat keresik. A keresés során különböző N táblaméretek esetén különböző mélységű kezdeményeket készítünk el. A következő táblázat foglalja össze, adott N értékek esetén, különböző *cut* mélységek esetén hány

(*db*) megoldáskezdemény található.

Az első tesztelés során különböző mennyiségű (4,8,16) feldolgozó doboz működési sebességét állítjuk szembe a szekvenciális változat sebességével. Választani kell, melyik N érték esetén hány megoldáskezdeménnyel dolgozunk. A működési sebességet ez jelentősen befolyásolni képes, tapasztalati úton igyekszünk [1000..4000] itervallumban tartani a kezdemények számát. Ez nem terheli túlzottan a hálózati kommunikációt, és kellő mennyiségű számolási feladatot hagy a feldolgozó dobozoknak. A táblázat vastaggal szedett értékei mutatják, mely N érték esetén mely felosztási mélységet választottunk (ez az érték egyben a 6.6. példában szereplő *cut* paraméter értéke is).

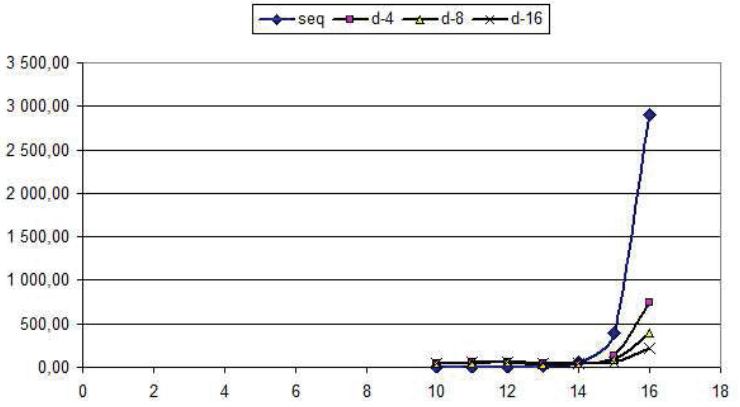
N	cut	db	cut	db	cut	db	cut	db
10	2	72	3	364	4	**1 400**	5	3 916
11	2	90	3	536	4	**2 468**	5	8 492
12	2	110	3	756	4	**4 080**	5	16 852
13	2	132	3	**1 020**	4	6 404	5	31 100
14	2	156	3	**1 364**	4	9 632	5	54 068
15	2	182	3	**1 764**	4	13 980	5	89 428
16	2	210	3	**2 236**	4	19688	5	141 812

A mérési eredmények az Eszterházy Károly Főiskola egyik géptermében végeztük, amely 19 db, hálózatba kötött NEC asztali számítógéppel van felszerelve. A gépek legfontosabb paraméterei: Intel i945G Chipset alaplap, 2 x 1GB RAM DDR2 / 667 memória, Intel Core 2 Duo E4500 2MB L2 2.2 GHz 800 MHz processzor, 160 GB S-ATA HDD, Integrated PCI Express Gigabit network, mely a régebbi típusú switch miatt csak 100 Mb/s sebességet biztosít. A gépeken Windows XP Service Pack 3 operációs rendszer fut, az ICE 3.2.1 verziójú változata biztosította a köztes réteg szolgáltatást.

N	cut	seq-idő	d-4	d-8	d-16
10	4	0,04	41,32	44,48	44,31
11	4	0,23	53,50	50,71	51,81
12	4	1,37	59,51	61,67	63,26
13	3	8,46	39,03	36,20	38,84
14	3	55,73	45,45	45,45	38,13
15	3	389,12	129,48	80,43	57,32
16	3	2 908,92	745,32	389,90	213,03

A táblázat *seq-idő* oszlopa mutatja az adott N érték és *cut* érték mellett mekkora volt a szekvenciális verzió futási ideje. Mint látszik, a futási idő az N-nel arányos módon exponenciálisan nő. A *d-4* oszlopban szerepelnek a 4, a *d-8* a 8, a *d-16* a 16 feldolgozó dobozzal történt elosztott működés futási idői. Az elosztott futási idők kis N értékek esetén nem változnak jelentősen, mivel a dobozok és a csatornák betöltése

és összekapcsolása futási időben jelentős *overhead*-et okoz. A N=12 és N=13 közötti váltásnál nem nőtt jelentősen a futási idő, ez annak köszönhető, hogy N=12 és cut=4 érték esetén 4080 megoldáskezdemény volt, míg N=13-nál a cut-ot lecsökkentettük 3 értéke, ami miatt a megoldáskezdemények száma is csökkent 1020-ra. Ezek után a futási idő újra elvárt módon emelkedni kezdett (6.7. ábra).

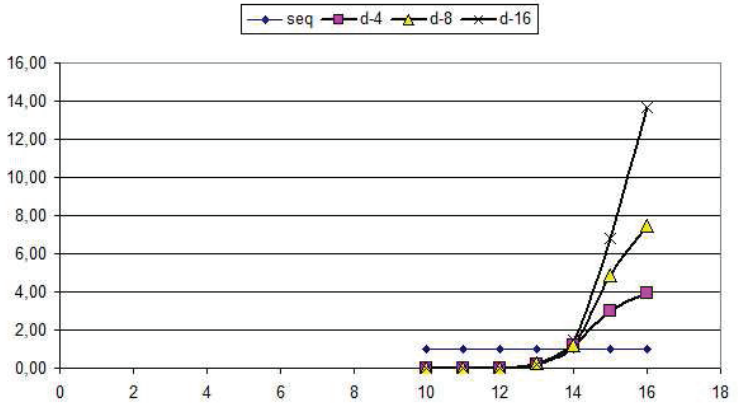


6.7. ábra. N Vezér futási idők

Az alábbi táblázat a szekvenciális futási idő értékét tekintve alapértéknek (1.0 érték) mutatja az elosztott változat futási idejének arányát:

N	cut	seq-idő	d-4	d-8	d-16
10	4	1,00	0,00097	0,00090	0,00090
11	4	1,00	0,00430	0,00454	0,00444
12	4	1,00	0,02302	0,02222	0,02166
13	3	1,00	0,22246	0,23370	0,21782
14	3	1,00	1,22618	1,22618	1,46177
15	3	1,00	3,00525	4,83800	6,78856
16	3	1,00	3,90291	7,46068	13,65523

A *d-4* oszlopbeli *speed-up* értékek növekvő N esetén (és növekvő futási idő esetén) közelednek a 4,0 értékhez, vagyis az elosztott rendszer futási ideje 4 feldolgozó doboz esetén közelít az elvárható maximális gyorsulási értékhez. Hasonlóan a *d-8* oszlop a 8-as, a *d-16* oszlop a 16-os gyorsulási értékhez közelít (6.8. ábra).



6.8. ábra. N Vezér gyorsulási értékek

C nyelvi kód

```
1 int pihenes(int x, int c1, int c2)
2 {
3    for (i=0;i<c1;i++)
4      for( j = 0; j < c2; j++)
5             if (j%1000==0) j=j;
6     return x;
7 }
```

6.9. példa. Lassító függvény

6.2. Nehezített művelet

A további tesztek során a visszalépéses keresés kovJoLepes függvényében használt *nincs_utesben* elemzést „nehezítjük” egy lassító függvény segítségével. A lassító függvény azt az esetleges valós életbeli működést helyettesíti, amikor a visszalépéses probléma esetén a kölcsönös kizárás ellenőrzése bonyolultabb műveletsor kiértékelését jelentené. A lassító függvényt C nyelven programoztuk le. Három paraméteres változata egymásba ágyazott ciklusok futása révén időigényes műveletsornak mutatja magát (6.9. példa).

A C nyelvi kódot .obj kiterjesztésű tárgykódú változatra fordítottuk le, majd csatoltuk a C linker számára szóló szerkesztési listába. A Clean nyelvi kódba pedig a függvény hívása került be (6.10. példa).

Clean nyelvi kód

```
1 pihenes::!Int !Int !Int -> Int
2 pihenes data num mult = code {
3     ccall pihenes "III:I"
4     }
```

6.10. példa. Lassító függvény Clean oldali interface

Clean nyelvi kód

```
1 nincs_utesben :: Vezer [Vezer] -> Bool
2 nincs_utesben _  []    = True
3 nincs_utesben r [h:tl]
4   | pihenes 1 slow_i slow_j == 0 = abort "slow"
5   | (r.x==h.x) = False
6   # dx = abs (r.x-h.x)
7   # dy = abs (r.y-h.y)
8   | (dx==dy)   = False
9   = nincs_utesben r tl
```

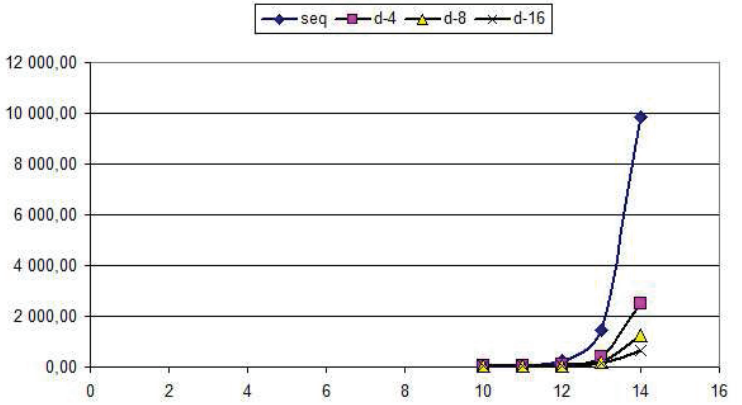
6.11. példa. Lassító függvény algoritmusba illesztése

A megfelelő ponton beépítettük a *nincs_utesben* függvénybe (6.11. példa 4. sor). A *slow_i* és *slow_j* értéke a programban konstansként kerültek be, a *slow_i=10*, *slow_j=100* értékekkel. Ez összesen 1000 ismétlést jelent. A *pihenes* függvény első paramétere 1, amely a C nyelvi változat visszatérési értéke is egyben. Ez azt jelenti, hogy az érték sosem lesz 0, vagyis az *abort "slow"* sosem hajtódik végre, ezzel teljes a Clean oldali hívás.

A lassítás beépítése drasztikusan növelte a futási időt. Emiatt az $N = 15$ és $N = 16$ értékek esetén mérésére nem került sor (az időigény a szekvenciális lefutás esetén körülbelül 170-szeresre nőtt, így $N = 15$ eset körülbelül 19 óra alatt futott volna le). Az alábbi táblázat mutatja a futási idők értékét (másodpercben):

N	slow	cut	seq-idő	d-4	d-8	d-16
10	10x100	4	6,64	46,38	44,15	46,21
11	10x100	4	37,59	52,88	47,15	51,01
12	10x100	4	228,17	89,90	64,89	60,25
13	10x100	3	1 456,14	400,34	218,89	126,67
14	10x100	3	9 867,29	2 518,32	1 274,75	662,53

A 6.12. grafikon alakja nagy hasonlóságot mutat a 6.7. grafikon alakjához. Mind a szekvenciális, mind az elosztott változat futási ideje a terheléssel arányos módon nőtt.



6.12. ábra. N Vezér gyorsulási értékek

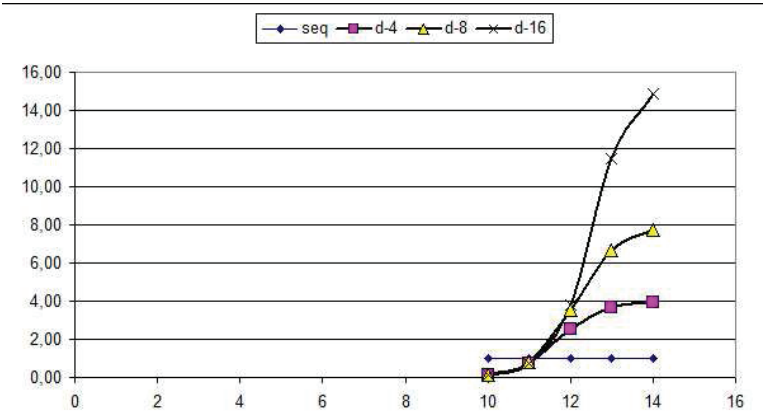
A gyorsulás értékek ezzel a módosítással jobban közelednek a maximális elvárható gyorsulási értékhez. Ennek oka, hogy a műveleti idő kisebb részét teszi ki a dobozok és a csatornák indításával járó előkészületi idő.

N	slow	cut	seq-idő	d-4	d-8	d-16
10	10x100	4	1,00	0,14	0,15	0,14
11	10x100	4	1,00	0,71	0,80	0,74
12	10x100	4	1,00	2,54	3,52	3,79
13	10x100	4	1,00	3,64	6,65	11,50
14	10x100	4	1,00	3,92	7,74	14,89

A gyorsulási grafikon a 6.13. ábrán látható, amennyiben 10×100 lassítási tényezőt alkalmazunk.

6.3. Vágási pont módosítása

Amennyiben más *cut* értéket használunk, úgy a futási idő jelentősen módosul. Ennek első oka, hogy a generáló doboz is ugyanazon lassítás mellett állítja elő a megoldáskezdeményeket. Az alábbi táblázat mutatja lassítás alkalmazásával a különböző *cut* értékek esetén hány darab megoldáskezdeményt lehet előállítani, és az előállításnak mekkora az időigénye (másodpercben):



6.13. ábra. N Vezér gyorsulási értékek lassítás mellett

N	cut	db	idő	cut	db	idő	cut	db	idő
10	5	3 916	0,28	4	1 400	0,06	3	364	0,00
11	5	8 492	0,54	4	2 468	0,09	3	536	0,00
12	5	16 852	0,96	4	4 080	0,14	3	756	0,01
13	5	31 100	1,67	4	6 404	0,21	3	1 020	0,03
14	5	54 068	2,71	4	9 632	0,31	3	1 364	0,01
15	5	89 428	4,25	4	13 980	0,43	3	1 764	0,03
16	5	141 812	6,45	4	19 688	0,59	3	2 236	0,03

Mint látható, a leghosszab előállítási idő sem haladja meg az 6,45 másodperces értéket, melyet lassítás nélkül egyébként 0,00 másodperc lett volna. Ez legfeljebb 6,45 másodperccel növelhetné meg a teljes futási időt, amely elhanyagolható. Tehát a *cut* módosítása ezen a módon nem befolyásolja jelentősen a futási időt.

Azonban jelentősebb változással jár eltérő *cut* érték esetén az előállítható megoldáskezdemények darabszámának drasztikus növekedése. Mint látható, $N = 14$ és $cut = 5$ esetén már 54068 darab ilyen kezdemény állítható elő. Amennyiben ezt mind átküldjük a csatornán a kezdő dobozból az elosztó dobozba, majd onnan tovább valamely feldolgozó dobozba, akkor emiatt megnőtt csatornaműveletek számával arányosan nő a futási idő.

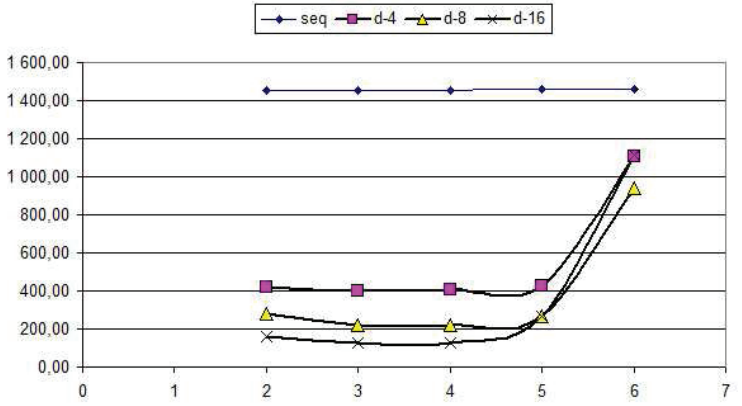
Másik fontos változás, hogy $N = 13$ és $cut = 3$ beállítás mellett a feldolgozó dobozoknak $13 - 3 = 10$ darab királynő elhelyezését kell elvégezni minden egyes megoldáskezdemény esetén, míg $cut = 5$ vágáskor már csak $13 - 5 = 8$ királynő marad hátra. Egy eset vizsgálatával tehát gyorsabban végeznek a feldolgozó dobozok. Egyszerű be-

látni, hogy növekvő *cut* érték mellett a teljes rendszer futási ideje közelít a szekvenciális megoldás futási idejéhez, a kommunikációs overhead miatt akár túl is léphetné azt.

Méréseket végeztünk $N = 13$ esetben különböző *cut* értékek használatával. A futási időt az alábbi táblázat foglalja össze:

N	cut	slow	seq	d-4	d-8	d-16
13	2	10x100	1 456,62	422,43	278,60	159,17
13	3	10x100	1 456,14	400,34	218,89	126,67
13	4	10x100	1 455,67	405,76	218,64	127,57
13	5	10x100	1 456,89	425,85	269,56	266,70
13	6	10x100	1 457,71	1 109,10	941,81	1 113,14

Mint látható, *cut* = 2,3,4 esetek mindegyike optimálisnak bizonyult az elosztott rendszer számára, a feldolgozó dobozok számától függetlenül. Az első esetben 1020, a második esetben 6404 megoldáskezdemény generálható. Arányosan nőtt a csatornakommunikáció mennyisége is, de ez nem befolyásolta számottevően a futási időt, az elosztó doboz kellő sebességgel tudta táplálni a feldolgozó dobozokat adatokkal. A *cut* értékének tovább növelésével azonban ugrásszerűen nőtt a megoldáskezdemények száma (*cut* = 5 esetben már 31100), és ezzel egyidőben csökkent az egyetlen eset feldolgozási időigénye, mivel egy vezérrel kevesebbet kellett a táblán elhelyezni. A feldolgozó dobozok feladatmegoldási idejének és a csatornakommunikációra fordított idejének aránya romlott. Ezt legjobban a 16 feldolgozó doboz érezte meg, melynél a leglátványosabb az időigény növekedése.



6.14. ábra. N Vezér változó *cut* értékekkel

6.4. Értékelés

A mérések során *splitf*, *split1*, *memory* output protokollok, *join1*, *joink*, *memory* input protokollok voltak a dobozokban használva. A csatornák *Int* és *Vezer* típusúak voltak. Ez utóbbi felhasználó által definiált rekord típus volt. A dobozokba kerülő kifejezések egyike, a *saver* rendelkezett előállítható **World* típusú paraméterrel is. A generált kód hibátlannak bizonyult ez esetekben, a rendszer hatékonysága a járulékos doboz és csatorna indítási műveletek és a csatornakezelési műveletek overhead-je mellett megközelítette az elvárható maximumot. A választott feladat számolás-intenzív volt, a kommunikációs költségek nem okoztak jelentős veszteséget. A *splitf* protokoll biztosította, hogy akár 16 feldolgozó dobozt is folyamatosan el tudjon látni feladattal egyetlen elosztó doboz. A futtató rendszer, és maga a D-Box koordinációs nyelv, a fordítóprogram a működőképességét ezzel a teszttel igazolta.

Összefoglaló

Az értekezésemben bemutatásra került egy koordinációs nyelv, melynek segítségével elosztott módon működő funkcionális programok kommunikációs gráfját lehet leírni. Ezen koordinációs nyelv segítségével statikus és dinamikus módon is felépíthető a gráf. A dinamikus eset képes alkalmazkodni a feldolgozandó adatok mennyiségbeli változásához. Definiálásra került a nyelv szintaktikája és statikus szemantikája is. Ezek segítségével a koordinációs nyelvi leírások helyessége fordítási időben, kódgenerálási fázis előtt leellenőrizhető. A statikus szemantika ellenőrzése kiterjed a számítási csomópontot leíró struktúra részei közötti összefüggések ellenőrzésére csakúgy, mint a csomópontokat összekötő élek, csatornák helyes használatának, címzésének ellenőrzésére. A dinamikus esetekben ezen felül az algráf példányok közötti kommunikációjának ellenőrzése is definiálásra került.

A koordinációs nyelven megadott gráf alapján sablonfájlokban tárolt forráskódok és makrók típussal paraméterezés révén kód generálható, mely már platform és programozási nyelv függő. A dolgozathoz mellékleteként csatolva van a Clean programozási nyelv, ICE köztes rétegre alapozott sablon gyűjtemény. Bemutatásra került a sablonok elkészítése során felhasználható makrók, valamint a futtató rendszer API függvényei, melynek segítségével más implementációs részletek, és más köztes réteghez is elkészíthetők a sablonok. A dolgozat tartalmaz egy példát, mely konkrét elosztott számítási feladatot valósít meg. A mérési elemzések és grafikonok bizonyítják, hogy a rendszer a valós életben is képes helytállni.

A saját eredmények a következők:

1. Létrehoztam egy olyan koordinációs nyelvet, amelynek segítségével leírható egy olyan számítási gráf, amelyben a számítási csomópontokon Clean funkcionális programozási nyelven megírt alkalmazások futnak. A koordinációs nyelv speciális nyelvi elemeket is támogat, mint például a `*World` típusú helyreállítható érték. Ezen típusú értékek csatornákon keresztül nem szállíthatóak, a fogadó oldalon egy helyettesítő érték generálható.

2. A koordinációs nyelv szintaktikai és statikus szemantikai szabályait bemutatattam, melynek alapján egy működő szintaktikai ellenőrző programot készítettem el. A dolgozathoz elkészült Windows operációs rendszer és ICE köztes rétegre

alapozott sablongyűjtemény, mely a DVD mellékleten csatolásra került. A sablonfájlok nevei és a könyvtárszerkezet egy XML konfigurációs fájlba került, mely alapján a kódgenerálási fázis jól paraméterezhetővé vált.

3. A futtatáshoz szükséges futtató rendszert, amely a választott köztes réteg szolgáltatásait egészíti ki a speciális igényeknek megfelelően, szintén bemutatattam. A dolgozat tartalmazza ezen futtató rendszer API függvényeinek leírását, valamint a futtató rendszer műveleti szemantikáját. A dolgozat mellékleteként egy .NET Frameworkben készített, ICE köztes rétegre alapozott futtató rendszert készítettem.

4. A fordítást, kódgenerálást és futást egy valós életbeli, számolásintezív probléma megvalósításával ellenőriztem. A generált kód szintaktikailag helyes volt, melyet az adott programozási nyelvek fordítóprogramjai a generált kód fordításával igazoltak. A szemantikai helyességet az elosztott projekt futásának eredményeként keletkezett kimeneti fájl tartalma igazolta. A mérési eredmények időtáblázatát, és grafikonjait megadtam, a mérési eredményeket kommentáltam.

A fenti eredmények közül a koordinációs nyelv statikus szemantikai elemzését, és a protokollok specifikációjának definiálását tartom a legfontosabbnak, erre fordítottam a legnagyobb hangsúlyt. A koordinációs nyelvet igyekeztem minél általánosabban elkészíteni, de nem hagyhattam figyelmen kívül, hogy egy konkrét, magasabb szintű leíró nyelvet (D-Clean) kellett támogatnom, ezért került a D-Box nyelvbe speciális Clean nyelvre jellemző elemek támogatása. A dolgozat melléktermé a futtató rendszer, melynek mélységi tárgyalását és specifikálását nem tekintettem elsődleges feladatnak. Ezért sem annak kidolgozottsági szintje, sem minőségi implementálása nem volt magas prioritású.

Conclusions

The present thesis will introduce a coordination language that allows the describing of the communication graph of distributed functional programmes. With the help of this coordination language the graph can be built up in a static and dynamic way as well. The dynamic case is able to adapt to the change of the amount of data to be processed. The syntax and static semantics of the language was also defined. With the help of all these the correctness of coordination language descriptions can be checked during translation, before the code-generation phase. The checking of static semantics covers the check of relationships between parts of the structure describing the computing nodes, as well as the check of proper use and addressing of channels connecting the nodes. In the dynamic cases the check of communication among the sub-graph instances was also defined.

On the basis of the graph given on a coordination language, a code can be generated through parameterizing with types the source code and macros stored in template files, and this is platform- and programming language dependant. Attached to the thesis you can find the template files written in Clean programming language, based on I.C.E. middleware. The macros applicable during the creation of templates and the API functions of the run-time system were introduced. With the help of these the templates can be created for different platforms and middleware. The thesis contains an example that implements a concrete distributed computation problem. Data analysis and charts have proven that the system is able to fulfil its task in real life as well.

The own results are as follows:

1. I have created a coordination language through which a kind of distributed computation graph can be defined in which applications written on a Clean functional programming language are running on the computing nodes. The coordination language supports special language elements as well, such as the `*World` type restorable value. These value types can not be carried through channels, a replacing value can be generated on the host side.

2. I have introduced the syntactical and static semantic rules of the coordination language, on the grounds of which I have created a working syntactical checking programme. Based on the Windows operation system and the ICE middleware

a template collection was also created and attached to the thesis as a DVD supplement. The names of template files and the library structure was put into an XML configuration file on the basis of which the code generating phase is more easy to parameter.

3. I have also introduced the essential run-time system, that completes the services of the chosen middleware according to the special requirements. The thesis contains the description of the API functions, and the operational semantics of the run-time system. As a supplement to the thesis I have created a working run-time system based on the ICE middleware, created in the .NET Framework.

4. Translation, code generation and running was checked through fulfilling a real-life, computation-intensive problem. The generated code was syntactically correct, underlined by the language compilers of the given programming language. The semantic correctness was underlined by the content of the output file created as a result of running the spaced project. The time table and charts of measured results are provided, the results are commented.

From among the above results I consider the semantic analysis of the coordination language and the punctual defining of the specifications of protocols the most important; the highest emphasis was placed on these. I strived at creating the coordination language as general as possible, though I could not ignore that I had to support a concrete, higher level descriptive language (D-Clean), that is why the support of special Clean language elements were included. The by-product of the thesis is the run-time system the deep description and specification of which was not considered as a primary task. That is why neither its level of elaboration, nor its qualitative implementation was of high priority.

Irodalomjegyzék

[1] *IBM developerWorks:Blogs:Bobby woolf:WebSphere SOA and J2EE in Practice* at `http://www.ibm.com/developerworks/blogs/page/woolf?entry=ibm_sequoia_supercomputer`

[2] Ken Arnold, James Gosling, David Holmes: *Java(TM) Programming Language*, Prentice Hall PTR; 3 edition (June 5, 2000), ISBN 978-0201704334

[3] Sun Microsystems, Inc.: *RFC1050 - RPC: Remote Procedure Call Protocol specification*, April 1988, at `http://www.faqs.org/rfcs/rfc1050.html`

[4] David S. Platt: *Introducing Microsoft©.NET* Microsoft Press, April 2003, ISBN: 9780735619180

[5] Matthew MacDonald: *Microsoft©.NET Distributed Applications: Integrating XML Web Services and .NET Remoting*, Microsoft Press (March 26, 2003) ISBN: 978-0735619333

[6] `http://www.mono-project.com/Main_Page`, (Mono is a cross platform, open source .NET development framework)

[7] *CommonObject Request Broker Architecture: Core Specification* December 2002 Version 3.0 - Editorial update

[8] William Gropp, Ewing Lusk, Anthony Skjellum: *Using MPI – Portable Parallel Programming with Message-Passing Interface* MIT Press, 1999

[9] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek,Vaidy Sunderam: *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing* , MIT Press, 1994, `http://www.netlib.org/pvm3/book/pvm-book.html`

[10] Kevin Hammond, Greg Michaelson, Robert Pointon: *The Hume Report, version 1.1*, `http://www-fp.cs.st-andrews.ac.uk/hume/report/`

[11] Jost Bertold: *Explicit and Implicit Parallel Functional Programming: Concepts and Implementation*, PhD Disszertáció, 2008, Marburg.

[12] Jones, S. P., Gordon, A., Finne, S.: Concurrent Haskell, Conference Record of POPL '96: The 23rd ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, Glasgow, 1996, 11 pp.

[13] Finne, S. and Jones, S., P. J.: Concurrent Haskell, In Principles Of Programming Languages, St. Petersburg Beach, Florida, 1996, pp. 295–308

[14] C´edric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 372-385, 1996.

[15] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1-40 and 41-77, 1992.

[16] Fournet, C., Le Fessant, F., Maranget, L., Schmitt, A.: *The JoCaml language beta release, Documentation and user's manual*, INRIA, 2001.

[17] Leroy X. et al. The Objective Caml Language (version 3.10). Software and documentation, available at `http://caml.inria.fr`, 2007.

[18] J. Barklund and R. Virding. Erlang Reference Manual, 1999. Available from `http://www.erlang.org/download/erl_spec47.ps.gz`. 2007.06.01

[19] Cole, M.: Algorithmic Skeletons, In: Hammond, K., Michaelson, G. (Eds.): Research Directions in Parallel Functional Programming, pp. 289-303, Springer-Verlag, 1999.

[20] Rabhi, F.A., Gorlatch, S. (Eds.): *Patterns and Skeletons for Parallel and Distributed Computing*, Springer Verlag, 2002.

[21] Jost Berthold, Ulrike Klusik, Rita Loogen, Steffen Priebe, and Nils Weskamp: *High-Level Process Control in Eden*, In: Kosch, H., Böszörményi L., Hellwagner H. (eds.): Parallel Processing, 9th International Euro-Par Conference, Euro-Par 2003, Proceedings, Klagenfurt, Austria, August 27-29, 2003, Springer Verlag, LNCS Vol. 2790, pp. 723-741.

[22] Ricardo Pena , Fernando Rubio , Clara Segura: *Deriving Non-Hierarchical Process Topologies*, Selected papers from the 3rd Scottish Functional Programming Workshop (SFP01), p.51-62, August 22-24, 2001

[23] Best, E., Hopkins, R. P.: B(PN)^2 - a Basic Petri Net Programming Notation, In: Bode, A., Reeve, M., Wolf, G. (Eds.): *Parallel Architectures and Languages Europe, 5th International PARLE Conference, PARLE'93*, Proceedings, Munich, Germany, June 14-17, 1993, Springer Verlag, LNCS Vol. 694, pp. 379-390.

[24] Horváth Z., Zsók V., Serrarens, P., Plasmeijer, R.: Parallel Elementwise Processable Func- tions in Concurrent Clean, Mathematical and Computer Modelling, No. 38, 2003, pp. 865-875.

[25] Kesseler, M.H.G.: *The Implementation of Functional Languages on Parallel Machines with Distributed Memory*, PhD Thesis, Catholic University of Nijmegen, 1996.

[26] Serrarens, P.R.: *Communication Issues in Distributed Functional Computing*, Ph.D. Thesis, University of Nijmegen, January 2001.

[27] Horváth Z., Zsók V., Serrarens, P., Plasmeijer, R.: *Parallel Elementwise Processable Functions in Concurrent Clean*, Mathematical and Computer Modelling 38, pp. 865-875, Pergamon, 2003.

[28] Hernyák Z., Horváth Z., Zsók V.: *Clean-CORBA Interface Supporting Skeletons*, Csőke Lajos(ed.) in.: Proceedings of 6th International Conference on Applied Informatics, Eger, Hungary, January 27-31, 2004. Eger, Hungary, B.V.B. Press, Vol. I. pp. 191-200.

[29] Fóthi Á., Horváth Z., Kozsik T.: Parallel Elementwise Processing – A Novel Version, In: Varga L., ed., *Proceedings of the Fourth Symposium on Programming Languages and Software Tools*, Visegrád, Hungary, June 9-10, 1995, pp. 180-194. and in *Annales Uni. Sci. Budapest de R. Eötvös Nom. Sectio Computatorica*, Vol. 17, pp. 105-124, 1998.

[30] Horváth Zoltán, Hernyák Zoltán, Zsók Viktóra: *Coordination Language for Distributed Clean*, Acta Cybernetica (ISSN: 0324-721 X), Vol. 17 (2), Institute of Informatics, University of Szeged, Szeged, Hungary, 2005, pp. 247-271. Selected publication of CSCS PhD Conference in Computer Science.

[31] Zsók V., Horváth Z., Varga Z.: *Functional Programs on Clusters* In: Striegnitz, Jörg; Davis, Kei (Eds.): Proceedings of the Workshop on Parallel/High- Performance Object-Oriented Scientific Computing (POOSC'03), Interner Bericht FZJ-ZAM-IB-2003-09, July 2003, pp. 93-100.

[32] Achten, P., Wierich, M.:A Tutorial to the Clean Object I/O Library, University of Nijmegen, 2000. `http://www.cs.kun.nl/~clean`

[33] Rinus Plasmeijer, Marko van Eekelen: Version 2.0 Language Report, University of Nijmegen, 2001. `http://clean.cs.ru.nl/download/Clean20/doc/CleanRep2.0.pdf`

[34] Plasmeijer,R.-van Eekelen,M.: Functional Programming and Parallel Graph Rewriting, Addison-Wesley, 1993.

[35] [EeNoPlSm90] van Eekelen,M. et al.: Concurrent Clean, Technical Report no 90-20, November 1990, University of Nijmegen.

[36] Plasmeijer, R., van Eekelen, M.: Concurrent Clean Language Report, University of Nijmegen, 2001.

[37] `http://www.mico.org/`

[38] `http://www.zeroc.com/`

[39] *Distributed Programming with Ice*, Michi Henning, Mark Spruiell With contributions by Dwayne Boone, Brent Eagles, Benoit Foucher, Marc Laukien, Matthew Newhook, Bernard Normier, `http://www.zeroc.com/download/Ice/3.3/Ice-3.3.0.pdf`

[40] *Formális nyelvek*, Bach István, Egyetemi tankönyv, Második, javított kiadás, TYPOTEX Kiadó, Budapest, 2002.

[41] *Modern Operating Systems (2nd Edition)*, by Andrew S. Tanenbaum, ISBN 0-13-092641-8

[42] Hoare, C. A. 1974. Monitors: an operating system structuring concept. Commun. ACM 17, 10 (Oct. 1974), 549-557.

[43] Andrew S. Tanenbaum , Maarten van Steen: *Elosztott rendszerek - alapelvek és paradigmák*, ISBN 976 545 387 6, Panem Kft, Budapest, 2004

A. függelék

Publikációs lista

Referált publikációk

1. Horváth Zoltán, *Hernyák Zoltán*, Zsók Viktóra: *Coordination Language for Distributed Clean*, Acta Cybernetica (ISSN: 0324-721 X), Vol. 17 (2), Institute of Informatics, University of Szeged, Szeged, Hungary, 2005, pp. 247-271. Selected publication of CSCS PhD Conference in Computer Science.

2. Horváth Zoltán, *Hernyák Zoltán*,Kozsik Tamás, Tejfel Máté, Ulbert Attila: *A Data Intensive Application on a Cluster - Parallel Elementwise Processing*, in P. Kacsuk, D. Kranzlmüller, Zs. Nemeth, J. Volkert (Eds.): Distributed and Parallel System - Cluster and Grid Computing, Proc. 4th Austrian-Hungarian Workshop on Distributed and Parallel Systems, Kluwer Academic Publishers, The Kluwer International Series in Engineering and Computer Science, Vol. 706, pp. 46-53, Linz, Austria, September 29-October 2, 2002.

3. Zsók Viktória, *Hernyák Zoltán*, Horváth Zoltán: *Designing Distributed Computational Skeletons in D-Clean and D-Box*, in.: Lecture Notes in Computer Science, Horváth Zoltán(ed.) in.: Central European Functional Programming School (The First Central European Summer School, CEFP 2005, Budapest, Hungary, July 4-15, 2005), Revised Selected Lectures. ISSN 0302-9743, vol. 4164, 2006, pp. 229-265.

4. Zsók Viktória, *Hernyák Zoltán*, Horváth Zoltán: *Distributed Pattern Design in D-Clean*, Central European Functional Programming School, CEFP 2005, ELTE, Budapest, Hungary, July 4-15, 2005, Lecture Notes, 33 pages

5. Zsók Viktória, *Hernyák Zoltán*, Horváth Zoltán: *Improving the Distributed Elementwise Processing Implementation in D-Clean*, In: Horváth Z., Kozma L, Zsók V. (eds): Proceedings of the 10th Symposium on Programming Languages and Software Tools (ISBN: 978-963-463-925-1), SPLST 2007, Dobogókő, Hungary, June 14-16, 2007, Eötvös University Press, 2007, pp. 256-264.

6. Zsók Viktória, *Hernyák Zoltán*, Horváth Zoltán: *Distributed Pattern Design in D-Clean*, Vene V., Meriste M.(ed.) in.: Proceedings of the Ninth Symposium on Programming Languages and Software Tools, ISBN: 9949-11-113-7, SPLST 2005, Tartu, Estonia, 13-14 August, 2005, Tartu University Press, 2005, pp. 220-234.

Referált kiadványokban megjelent publikációk

7. Zsók Viktória, *Hernyák Zoltán*, Horváth Zoltán: */Distributed Computation on Cluster using D-Clean and D-Box. Extended abstract* In: Davis, K., Quintino, T., Striegnitz, J. (eds): 5th Workshop on Parallel/High Performance Object-Oriented Scientific Computing, POOSC'06 at 20th European Conference on Object-Oriented Programming, ECOOP 2006, Nantes, France, 3rd July, 2006, 3 pages. Summary: Object-Oriented Technology, ECOOP 2006 Workshop Reader, ECOOP 2006 Workshops, Nantes, France, July 3-7, 2006, Final Reports, LNCS 4379, Springer Verlag, 2007, pp. 141-145.

8. Horváth Zoltán, *Hernyák Zoltán*, Zsók Viktória: *Implementing Distributed Skeletons using D-Clean and D-Box*, In: Butterfield, A. (ed): Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages, IFL 2005, Dublin, Ireland, September 19-21, 2005, pp. 1-16.

9. *Hernyák Zoltán*, Horváth Zoltán, Zsók Viktória: *Clean-CORBA Interface Supporting Pipeline Skeleton*, Csőke Lajos(ed.) in.: Proceedings of 6th International Conference on Applied Informatics, Eger, Hungary, January 27-31, 2004. Eger, Hungary, B.V.B. Press, Vol. I. pp. 191-200.

Nemzetközi konferencia kiadványokban megjelent publikációk

10. Zsók Viktória, Horváth Zoltán, *Hernyák Zoltán*: */Distributed Elementwise Processing in D-Clean*, In: Nilsson, H. (ed): Proceedings of the Seventh Symposium on Trends in Functional Programming, TFP 2006, Nottingham, UK, 19-21 April, 2006, The University of Nottingham, pp. 378-386.

11. *Hernyák Zoltán*, Horváth Zoltán, Zsók Viktória: *Design of Language Elements for Dynamic Distributed Computation of Clean Expressions on Clusters*, in: Loidl, H-W. (ed): Proceedings of Fifth Symposium on Trends in Functional Programming, TFP 2004, Munich, Germany, November 25-26, 2004, Ludwig-Maximilians University, pp. 257-270.

12. *Hernyák Zoltán*: *PEDPI as a Message Passing Interface with OO support*, in: Striegnitz, Jörg; Davis, Kei (Eds.) (2003) Proceedings of the Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'03), Interner Bericht FZJ-ZAM-IB-2003-09, Juli 2003, pp. 93-100.

Egyéb publikációk, tevékenységek

13. Roland Király, *Zoltán Hernyák. Elosztott rendszerekre implementált funkcionális nyelvek - PLMR projekt*, Networkshop 2008

14. *Hernyák Zoltán*, Horváth Zoltán, Zsók Viktória: *Implementing Pipeline Skeleton in Clean using CORBA Channels. Position paper*, Proceedings of the Workshop on Multiparadigm Programming with Object-Oriented languages, MPOOL 2004, Workshop #11 at 18th European Conference on Object-Oriented Programming, ECOOP 2004, University of Oslo, Oslo, Norway, June 15, 2004, 6 pages. (Position paper version of 'Clean-CORBA Interface Supporting Pipeline Skeleton')

15. Zsók Viktória, Horváth Zoltán, *Hernyák Zoltán Distributed Elementwise Processing in D-Clean*, in 2nd Central-European Functional Programming School, CEFP 2007, Cluj-Napoca, Romania, PhD Workshop, Abstracts, Babeş-Bolyai University, 2 pages.

16. *Hernyák Zoltán*, Horváth Zoltán, Zsók Viktória *Implementing a Distributed Skeleton in Clean. Abstract*, In: Horváth Z., Zsók V. (eds): Proceedings of the 18th International Symposium on Implementation and Application of Functional Programming Languages, ELTE IK TR 2006-S01, ISBN 963-463-876-7, 2006, pp. 145.

17. Emőd Kovács, *Zoltán Hernyák*, Tibor Radványi, Roland Király: *A C# programozási nyelv a felsőoktatásban* Programozási tankönyv. Felsőoktatási jegyzet. 2005. (published in digital format only)

18. Emőd Kovács, *Zoltán Hernyák*, Tibor Radványi, Roland Király: *A C# programozási nyelv a felsőoktatásban* „Algorithms of Informatics" kiadványban. ELTE Faculty of Informatics, 2005 (published in digital format only)

19. *Hernyák Zoltán*, Horváth Zoltán, Zsók Viktória *Control Language for Distributed Clean*, in Conference of PhD Students in Computer Science, Volume of Extended Abstracts, Szeged, Hungary, July 1-4, 2004, pp. 54.

20. *Hernyák Zoltán*: *Elosztott programozás oktatása a gyakorlatban*, Informatika a felsőoktatásban 2002, 2002 aug.25-aug.30, Debrecen

21. 2003.10.01-2004.01.31, *CEEPUS Scholarship for PhD students*, Linz, Johannes Kepler University.

B. függelék

D-Box nyelv EBNF szintaktikai leírása

Az alábbiakban olvasható a *D-Box* nyelv szintaktikai leírása EBNF formában. A szabályok nevei és értékei között a szabály-definiáló értékadás jelét használjuk, a `:==` jelet. A szabályokat . (pont) zárja. A terminális szimbólumok aposztrófok között szerepelnek.

A *D-Box* nyelv a C nyelvhez hasonlóan erősen épít a kapcsos zárójelpárokra. Az összetartozó részeket ilyen kapcsos zárójelpárok, az alrészeket gyakran gömbölyű zárójelpárok fogják össze. A felsorolásokat vesszővel választjuk el egymástól.

A *D-Box* definíciókba épített Clean nyelvi kifejezéseket a határoló aposztróf (Alt-7, `‘kifejezés‘`) veszi körbe. Ez a karakter nem fordulhat elő a Clean nyelvi szintaktikai szerint a kifejezésekben, ezért alkalmas a feladatára.

Engedélyezett a C stílusú egysoros komment írása, // karakterekkel kezdve. Az első *start* szabály szerint a forráskódban csak típusdefiníciók és doboz definíciók szerepelhetnek, tetszőleges sorrendben.

```
start      :==  definition | definition start.
definition :==  typedef | boxDef.
```

B.1. Típus-definíció leírása a *typedef* segítségével

```
typedef    :==  "typeDef" identifier "=" "{" fieldTypeL "}".
fieldType  :==  "Real" | "Int" | "Bool" | "Char" | identifier.
fieldTypeL :==  fieldType | fieldType "," fieldTypeL.
```

B.2. Doboz-definíció leírása a *box* segítségével

```
boxDef     :==  "BOX" BoxID "{" boxBody "}".
BoxID      :==  "BoxID_"intNumber.
boxBody    :==  intNumber "," inpProt "," exprDef "," OutProt.
```

B.2.1. Az input rész leírása

```
inpProt      :==  "{" "(" IChannelDefL ")" "," IProtMain "}".
IProtMain    :==  "join1" | "joink" | "memory".

IChannelDef  :==  INull | IFix | IAutoConnBox.
IChannelDefL :==  IChannelDef | IChannelDefL "," IChannelDef.
INull        :==  "(" "null" ")".

IFix         :==  "(" CType "," intNumber ")".
IAutoConnBox :==  "(" intNumber "," BoxID "," "(" CTypeL ")" ")".
BType        :==  fieldType.

CType        :==  BType | "[" BType "]" | "[[" BType "]]".
CTypeL       :==  CType | CType "," CTypeL.
EType        :==  CType | "null" | "[[[" BType "]]]" | "*World";
ETypeL       :==  EType | EType "," ETypeL.
```

B.2.2. A kifejezés leírása

```
exprDef  :==  "{" "(" ETypeL ")" "," CleanFv "," "(" ETypeL ")" "}".
CleanFv  :==  " String ".
```

B.1. megjegyzés. A $\langle CleanFv \rangle$ egy Clean nyelvi kifejezést takar. Ezen kifejezés szintaktikai leírása túlmutat ezen dolgozat keretein. A Clean nyelvi kifejezések szintaktikai leírását a Clean Language Manual [33] tárgyalja.

B.2.3. Az output rész leírása

```
OutProt      :==  "{" "(" OChannelDefL ")" "," OProtMain "}".
OProtMain    :==  "split1" | "splitk" | "splitf" intNumber | "memory";
OChannelDef  :==  ONull | OFix | OAuto | OConnBox | OStartGraph.
OChannelDefL :==  OChannelDef | OChannelDefL "," OChannelDef.
ONull        :==  "(" "null" ")".

OFix         :==  "(" CType "," intNumber ")".
OAuto        :==  "(" CType "," "auto" ")".
OConnBox     :==  "(" "connBox" threadID BoxID intNumber
                  "(" CTypeL ")" ")".
OStartGraph  :==  "(" "startGraph" intNumber NumExpr BoxID
                  "(" CTypeL ")" ")".
NumExpr      :==  " String ".
threadID     :==  "ancestorThread" | "thisThread".
```

B.2. megjegyzés. A $\langle NumExpr \rangle$ egy olyan Clean nyelvi kifejezés, melynek eredménye egy egész szám. Ezen kifejezés szintaktikai leírása túlmutat az értekezés keretein. A Clean nyelvi kifejezések szintaktikai leírását a Clean Language Manual [33] tárgyalja.

B.3. Yacc definíció

```
%token intNumber
%token identifier equation
%token BoxID
%token k_typeDef

%token k_Int k_Real k_Bool k_Char
%token k_Left k_Right k_LeftC k_RightC
%token k_comma
%token k_Box
%token k_join1 k_joink k_memory k_null
%token k_auto
%token k_World
%token k_left_L k_left_LL k_left_LLL k_right_L k_right_LL k_right_LLL
%token k_split1 k_splitk k_splitf
%token k_connBox k_startGraph
%token k_ancestorThread k_thisThread
%token k_CleanFv k_graphID

%{
    #include <stdio.h>
    #include <stdlib.h>
    int yyerror(char *s);
    int yylex(void);
    int yytext;
%}
```

```
%%
start:        definition | definition start;

definition: typedef|boxDef;

typedef:    k_typeDef identifier equation
              k_Left fieldTypeL k_Right;

fieldType:  k_Real | k_Int | k_Bool | k_Char | identifier;
fieldTypeL: fieldType |fieldType k_comma fieldTypeL ;


boxDef:  k_Box BoxID k_Left boxBody k_Right;
boxBody: intNumber k_comma inpProt k_comma exprDef k_comma OutProt;
inpProt: k_Left k_LeftC IChannelDefL k_RightC k_comma IProtMain k_Right;

IProtMain:    k_join1 | k_joink | k_memory;
IChannelDef:  INull   | IFix    | IAutoConnBox;
IChannelDefL: IChannelDef | IChannelDefL k_comma IChannelDef;

INull:            k_LeftC k_null k_RightC;
IFix:             k_LeftC  CType  k_comma  intNumber  k_RightC;
IAutoConnBox:     k_LeftC  intNumber  k_comma  BoxID k_comma
                  k_LeftC CTypeL k_RightC k_RightC;

BType:    identifier  | k_Real | k_Int | k_Bool | k_Char;
CType:    BType | k_left_L BType k_right_L | k_left_LL BType k_right_LL;
CTypeL:   CType | CType  k_comma  CTypeL;
EType:    CType | k_null | k_left_LLL BType  k_right_LLL | k_World;
ETypeL:   EType | EType  k_comma  ETypeL;

exprDef:  k_Left
             k_LeftC  ETypeL  k_RightC k_comma
             ExpressionDef k_comma
             k_LeftC  ETypeL  k_RightC
          k_Right;

ExpressionDef: k_CleanFv;
```

```
OutProt:      k_Left
                k_LeftC OChannelDefL  k_RightC k_comma
                OProtMain
              k_Right;

OProtMain:    k_split1 | k_splitk | k_splitf intNumber | k_memory;

OChannelDef:  ONull | OFix | OAuto | OConnBox | OStartGraph;
OChannelDefL: OChannelDef | OChannelDefL k_comma OChannelDef;

ONull:    k_LeftC k_null k_RightC;
OFix:     k_LeftC  CType  k_comma  intNumber  k_RightC;
OAuto:    k_LeftC  CType  k_comma k_auto k_RightC;
OConnBox: k_LeftC k_connBox  threadID  BoxID  intNumber
            k_LeftC CTypeL  k_RightC
          k_RightC;

OStartGraph: k_LeftC k_startGraph  intNumber  k_CleanFv  BoxID
               k_LeftC CTypeL k_RightC
             k_RightC;

threadID:    k_ancestorThread | k_thisThread;

%%
int yyerror(char *s)
{
  printf("#error %s#\n", s);
        return 0;
}

int main ( int argc, char *argv[] )
{
    extern FILE *yyin;
    ++argv;--argc;
    yyin=fopen( argv[0], "r");
    yyparse();
    printf("Parse Completed\n");
    return 0;
}
```

B.4. Lex definíció

```
%{
    #include <stdlib.h>
    #include <stdio.h>
          #include "y.tab.h"
    int yyerror(char *);
%}
%%

"//".*\r*\n       ;
\n                ;
[\ \t\r]+         ;
[0-9]+            {printf("%s ",yytext); return intNumber;}

Int               {printf("%s ",yytext); return k_Int;}
Real              {printf("%s ",yytext); return k_Real;}
Bool              {printf("%s ",yytext); return k_Bool;}
Char              {printf("%s ",yytext); return k_Char;}

"{"               {printf("%s ",yytext); return k_Left;}
"}"               {printf("%s ",yytext); return k_Right;}
"("               {printf("%s ",yytext); return k_LeftC;}
")"               {printf("%s ",yytext); return k_RightC;}
,                 {printf("%s ",yytext); return k_comma;}
=                 {printf("%s ",yytext); return equation;}

BoxID_[0-9]+      {printf("%s ",yytext); return BoxID;}
BOX               {printf("\n%s ",yytext); return k_Box;}
typeDef           {printf("\n%s ",yytext); return k_typeDef;}
```

```
join1     {printf("%s ",yytext); return k_join1;}
joink     {printf("%s ",yytext); return k_joink;}
memory    {printf("%s ",yytext); return k_memory;}

null      {printf("%s ",yytext); return k_null;}
auto      {printf("%s ",yytext); return k_auto;}
"*World"  {printf("%s ",yytext); return k_World;}

"["       {printf("%s ",yytext); return k_left_L;}
"[["      {printf("%s ",yytext); return k_left_LL;}
"[[["     {printf("%s ",yytext); return k_left_LLL;}
"]"       {printf("%s ",yytext); return k_right_L;}
"]]"      {printf("%s ",yytext); return k_right_LL;}
"]]]"     {printf("%s ",yytext); return k_right_LLL;}

split1    {printf("%s ",yytext); return k_split1;}
splitk    {printf("%s ",yytext); return k_splitk;}
splitf    {printf("%s ",yytext); return k_splitf;}

connBox         {printf("%s ",yytext); return k_connBox;}
startGraph      {printf("%s ",yytext); return k_startGraph;}

ancestorThread  {printf("%s ",yytext); return k_ancestorThread;}
thisThread      {printf("%s ",yytext); return k_thisThread;}

[A-Za-z0-9_]*   {printf("%s ",yytext); return identifier;}
‘.*‘            {printf("[%s] ",yytext); return k_CleanFv;}

.               {yyerror ("unexpected character");}

%%
int yywrap(void) {
   return 1;
}
```

C. függelék

Futtató rendszer API ismertetése

A függvények paramétertípusainak megadása során típusnevekre hivatkozunk, melyek SLICE nyelvi definíciói az alábbiak:

```
#define TProxyDescr     string
#define TProjectID      string
#define TBoxID          string
#define TTypeOfChannel  string
#define TPath           string
#define TPlatform       string
#define ServerID        string
#define TThreadID       int
#define TChannelID      int
#define TChannelTask    int
#define TSubGraphID     int
#define TSubGraphTask   int
#define TErrCode        int
```

C.1. Általános megjegyzések

Egy számítási csomópontot három adat jellemez: *projectID*, *threadID*, *boxID*. Egy csatornát elvileg kettő adat jellemez (*projectID*, *channelID*), de csatornákkal kapcsolatos műveletek esetén mindig meg kell adni a csatorna típusát is, hogy egyértelmű legyen, hogy a műveletvégző is ismeri a típust.

A csatornák és egyéb szerver jellegű alkalmazások elérhetőségét a *TProxyDescr* leíró formájában adjuk meg. Ez egy több részből álló string, amely tartalmazza a protokoll nevét (pl. *TCP/IP*), az IP címet és egy portot.

A csatornák szerepkörét (*channelTask*) egy egész szám jellemzi. A 0 érték képviseli az *input* csatorna szerepkörét, az 1 érték az *output* szerepkört. Egyes esetekben a csatornák *kollekcióját* is meg kell adni. A *startGraph* csatornadefiniáló kifejezés

több példányban is képes indítani algráfokat (N példány), melyek kimenő csatornái ugyanazon doboz input csatornáira lesznek rákötve (*autoConnBox* csatornadefiniáló kifejezés). Ezen doboz az input csatornáinak sorozatát N példányban (kollekcióban) indítja el. Az azonos kollekcióba sorolt csatornák ugyanazon algráf kimenő csatornáira vannak rákötve, hogy ne keveredjenek össze az adatok. Amennyiben a csatornák nem *autoConnBox* alapján kerültek indításra, úgy a 0 kollekcióba tartoznak (a kollekciók sorszámozása 0-tól indul).

Az algráfok szerepkörét (*subGraphTask*) is egy egész szám írja le. Ez későbbi fejlesztési célok végett van jelenleg a rendszerben. A beágyazott D-Clean kifejezések indíthatnak speciális, hurok jellegű algráfokat, melyek szerepkörét meg kell (majd) különböztetni a *startGraph* csatornadefiniáló kifejezések által indítottaktól. Ez utóbbi jellegű algráfok *subGraphTask* értéke 0 kell legyen

Egyes függvények *TErrCode* jellegű értékkel térnek vissza. Ez egy speciálisan felépített string, mely három részből áll: „<flag>:<hibakod>|<hibauzenet>”. A flag vagy `-err` vagy `+ok` lehet, így egyértelműen azonnal eldönthető, hogy hibát jelez-e a visszatérési érték vagy sem. Amennyiben nem, úgy ezen string a `+ok:0|Successfull.` tartalmat hordozza. Egyéb esetekben a `-err:<hibakod>|<hibaüzenet>` alakú.

C.2. Code Library

A D-Box projektek futtatása előtt a kódot fel kell tölteni egy kódtároló szolgáltatásra. A *Code Library* szolgáltatásból egy alhálózatban egy is elegendő. Ezen szolgáltatás az adott gépen nem generál jelentős processzor és memória terhelést, passzív szerepkörű. Az adott számítógép diszkjén egy kijelölt alkönyvtárban tárolja a hozzá feltöltött bináris állományokat, annak megjelölésével, hogy melyik projekt milyen jellegű elemérő van szó, és az melyik platformra készült. A telepítési és beállítási egyszerűsítések végett ezeket az információkat nem adatbázisban őrzi, hanem a diszken eleve olyan alkönyvtárszerkezetet alakít ki, amelyből a fenti információkat képes utólag (akár gép újraindítás után is) újra kinyerni.

Az API függvényekben itt is, később is többször előfordul a *platform neve*. Ez egyszerű string, pl. *windows* vagy *linux*. További részletezés esetén pl. *windows_msvc* jelölheti a *windows* operációs rendszerű, Microsoft Visual C alapú alkalmazást. Az API függvények ezen plusz (meta jellegű) információt különösebben nem ellenőrzik, és nem korlátozzák a platform névválasztást.

A platform nevének választása akkor lesz majd fontos, amikor a futtatás során az *AppStarter* ütemező lekérdezi az elérhető node-ok platform kapacitását. A beérkező válaszok szintén string alakban adják meg az adott gép jellemzőit. Az AppStarter mindössze összehasonlítja a gépektől érkező választ a tárolt bináris fájloknál megadott

platformleírásokkal, és egyezőséget keres különösebb értelem nélkül. Vagyis, ha a feltöltésnél *windows_msvc* platformnévvel töltöttünk fel egy kódot, egy node pedig magát *windows* platformmal írja le, akkor nincs egyezés. Ugyanakkor egy rosszul beállított linux node is küldhet magáról *windows_msvc* leírást, amely ugyan egyezést mutat, működés közben valószínűleg nem fog. Ezért a platformleírásokat a rendszerben részt vevő gépek és rendszer-adminisztrátoroknak egyeztetniük kell, hogy konzenszus legyen a rendszerben.

> *bool CLEmptyProject(TProjectID projectID, bool Overwrite);*

A függvény meghívásával a paraméterben adott projekthez tartozó alkönyvtárszerkezetet felkészíthetjük a feltöltéshez. Az *overwrite* paraméterrel szabályozhatjuk, hogy viselkedjen a felkészítés, ha már az adott projekt valamely verziója létezik a szerveren:

- *overwrite=true* esetén a korábbi verzió (ha van) törlődik, és a függvény mindenképpen *true* (felkészülés sikeres) visszatérési értékkel tér vissza
- *overwrite=false* esetén amennyiben ilyen projekt névvel már létezett volna állomány a szerveren, úgy azok nem törlődnek, és a függvény *false* (nem sikeres) értékkel tér vissza. Ha ilyen projektazonosító még nem volt a szerveren, úgy az alkönyvtárszerkezetben a dobozok és csatornák későbbi tárolására felkészülvén létrejönnek, és a függvény *true* értékkel tér vissza.

> *bool CLUploadNode(TProjectID projectID, TSubGraphID subGraphID, TBoxID boxID, ByteStream file, TPath path, TPlatform platform);*

A függvény segítségével lehet valamely számítási doboz definícióhoz tartozó fájlokat feltölteni. A feltöltés során meg kell adni a projekt azonosítót, azon belül az algráf és a doboz azonosítót is. Ezen felül specifikálni kell a platform nevét, valamint a feltöltendő fájl nevét. A fájl tartalmát egy *byte-stream* írja le. A függvény a fenti információk alapján fájlt hoz létre a diszken, és a bájtsorozatot beleírja. A feltöltés sikerességét logikai értékként adja meg.

A feltöltés során egy kiinduló alkönyvtárat vesz alapul, pl `c:\app_env`. A cél fájlt az alábbi elérési útvonalú alkönyvtárba menti el: `kiinduló−könyvtár/projectID/subGraphID/boxID/platform/<file>`.

> *bool CLUploadChannel(TProjectID projectID, TTypeOfChannel typeOfChannel, ByteStream file, TPath path, TPlatform platform);*

Hasonlóan a *CLUploadNode* függvényhez: csatorna kód feltöltését teszi lehetővé. A csatornáknál meg kell adni a projektazonosítót, a csatorna típusának nevét, a platform nevét, a fájl nevét, és a bináris bájtsorozatot: a fájl tartalmát. A mentés a `kiinduló−−könyvtár/projectID/csatorna−típus/platform/file` fájlba történik.

> *int CLFileCountNode(TProjectID projectID, TSubGraphID subGraphID,*
> *TBoxID boxID, TPlatform Platform);*

Ez a függvény megszámolja, hogy egy adott projektben szereplő, adott algráfba sorolt adott platformra, adott doboz esetén hány fájlt kell letölteni a futtatáshoz (összesen hány fájl van tárolva). Egy doboz windows platformon ICE alapokon általában 2 fájl, egy *.exe*, és egy *.manifest* fájl, mely utóbbi a Microsoft Visual C++ library helyes működéséhez (run-time library illesztés) szükséges.

> *int CLFileCountChannel(TProjectID projectID, TTypeOfChannel typeOfChannel,*
> *TPlatform Platform);*

A függvény a *CLFileCountNode* függvényhez hasonlóan megadja, hogy egy adott projektben definiált adott típusú, adott platformon működő csatorna hány fájlból áll. Egy windows platformon ICE alapokon általában egy csatorna egy darab *.exe* fájlból áll (az icecs.dll a *gacutil* segítségével a gépen már regisztrálásra került).

> *ByteStream CLDownloadNode(TProjectID projectID, TSubGraphID subGraphID,*
> *TBoxID boxID, TPlatform Platform,int fileID, out string fileName);*

Ez a függvény adott projekt, adott algráf, platform és doboz azonosító, adott fájlsorszám esetén megadja az $fileID$ sorszámú fájl nevét (*fileName*), valamint magát a fájl tartalmát. A tárolt fájlok számát a *CLmFileCounNode*) függvény segítségével lehet lekérni. Amennyiben pl. 3-t ad meg (3 db fájl szükséges a futtatáshoz), úgy a 3 fájlt három *CLDownloadNode* függvényhívással lehet letölteni. A fájlok sorszámozása 0-tól indul. Amennyiben nem létező sorszámra hivatkozunk, úgy mind a fájlnév mind a bájt tömb *null* értékű lesz.

> *ByteStream CLDownloadChannel(TProjectID projectID,*
> *TTypeOfChannel typeOfChannel, TPlatform Platform, int fileID,*
> *out string fileName);*

A függvény a *CLDownloadNode* függvényhez hasonlóan működik, adott projektben definiált adott típusú, adott platformú csatorna kiépítéséhez szükséges valamelyik fájlt képes letölteni. A fájlok sorszámozása 0-tól indul. A függvény megadja a fájl nevét, és a fájl tartalmát egyidőben. Amennyiben nem létező sorszámra hivatkozunk, úgy *null* értéket ad meg a fájl neveként és tartalmaként.

> *ArrayOfTBoxID CLGetAllBoxID(TProjectID projectID, TSubGraphID subGraphID);*

A függvény megadja, hogy egy adott projekt és algráf milyen dobozdefiníciókat tartalmaz. A függvény visszatérési értéke ezen dobozazonosítók tömbje. Amennyiben ismeretlen projektazonosítót vagy algráf azonosítót adtunk volna meg, úgy a tömb nulla elemű lesz.

C.3. Application Starter

Ezen szerver az ütemező, melynek indítási feladatokat kell ellátni. Az indítási kérelem elsősorban a felhasználótól érkezik, mely projekt szintű indítást indukál. A projekt indítása az 1-es algráf dobozainak indítását jelenti. Ezek indítása azonban nem elég, futás közben dinamikus algráfindítási parancsokat is fel kell dolgozni, csakúgy, mint az elindult dobozok csatornáinak utólagos indítását.

> *string ASStartProject(TProjectID projectID);*

A függvény egy adott projekt indítását kezdeményezi. A sikeres indítást a következő visszatérési érték jelzi: `+ok:0|Successfull.` Ezen függvény hívása egyébként egyezik az `ASSubGraphStart(projectID,0,"<fake>",1,1,out threads);` függvényhívással.

> *string ASSubGraphStart(TProjectID projectID, TThreadID starterThread,*
> *TBoxID starterBoxID, TSubGraphID subgrapToStart,*
> *int count, out ArrayOfThreadID threadIDs);*

Ezen függvény az egyik legösszetettebb, fő feladata egy adott algráfba tartozó dobozok indítása a felügyelete alatt álló node-okon, megpóbálván a terheléseket egyenletesen elosztani. Az indítási paraméterek az alábbiak:

- *projectID* és a *subgrapToStart* azonosítja az indítandó dobozokat (ennyi információ alapján a megfelelő *CodeLib* szerver már meg tudja adni a dobozok listáját,
- a *starterThread* és a *starterBoxID* azonosítja az indítást kérő dobozt,
- a *count* adja meg hányszor kell indítani az algráfot.

A függvény visszatérési értéke egy hibaüzenet szöveges alakban. Ezen kívül generálja még az indított algráfok szálazonosítóit (*threadIDs* kimenő paraméter). Amennyiben sikertelen az indítás, a kimenő *threadIDs* vektor akkor is *count* elemű lesz, de -1 értékkel lesz feltöltve.

A lépések, amit a függvény végrehajt:

- *1. lépés*: begyűjti az elérhető *CodeLib* szerverek listáját,
- *2. lépés*: lekéri az adott projekt adott algráfjához tartozó dobozazonosítókat,
- *3. lépés*: lekéri melyik doboz milyen platformokon érhető el a különböző *CodeLib* szervereken,
- *4. lépés*: lekéri az elérhető *LocalComm* szerverektől a rajtuk futtatható platformok neveit és aktuális terhelési értékeiket,

- *5. lépés*: a kapott listákat összeveti, és futtatási tervet készít, összepárosítva elsősorban a platformokat, amennyiben több lehetőség is van, a kisebb terheltségű *LocalComm* szervert választva,
- *6. lépés*: új, egyedi szálazonosítókat kér le a *RegCenter*-től,
- *7. lépés*: a kiválasztott *LocalComm* szervereket felkéri a számukra kiválasztott doboz letöltésére és indítására.

```
TErrCode ASChannelStart( TProjectID projectID, TTypeOfChannel typeOfChannel,
    TChannelID suggestedChannelID, int collection, out TChannelID newChannelID );
```

Egy konkrét csatornát indít el. A csatornát azonosítja a *projectID* és a *typeOfChannel*. Az újonnan indított csatorna egyedi azonosítója automatikusan lesz generálva, ha a *suggestedChannelID*= 0, különben a megadott érték lesz használva. A *collection* adja meg, hogy a csatorna melyik kollekcióba tartozik.

Ezen kívül generálja még az indított csatorna egyedi azonosítóját (*newChannelID*), amely egyezik a *suggestedChannelID*-vel, ha az nem nulla, egyébként a generált értéket veszi fel.

C.4. Registration Center

A *Registration Center* (röviden *RegCenter*) a futtató rendszer névszolgáltatója, és egyéb szempontok szerint is információs központja. Mivel minden elindított doboz és csatorna ide regisztrálja be nevét és elérhetőségét, új csatornaazonosítók és szálazonosítók generálása is ide tartozó feladat.

```
int RCCreateChannelID(TProjectID projectID);
```

A függvény a megadott (futó) projektazonosítóhoz generál egy új, egyedi csatornaazonosítót. Ilyen azonosítókra a nem fix csatornaazonosítóval rendelkező csatornák számára van szükség.

```
int RCCreateThreadID(TProjectID projectID);
```

A függvény a megadott (futó) projekt azonosító alapján generál egy új, egyedi szálazonosítót. Minden indított algráf példánynak (az 1-es, vezérlő gráf példánynak is) szüksége van szálazonosítóra.

```
TErrCode RCRegisterBox(TProjectID projectID,TBoxID boxID,
        TThreadID boxThreadID, TThreadID starterThreadID, TBoxID starterBoxID,
        string localCommProxy);
```

A függvény segítségével lehet egy elindult doboznak jelezni a lokációját. A doboz azonosítja magát a szokásos hármassal, majd megadja, őt ki indította el az *starterThreadID*, *starterBoxID* alapján, és megadja az őt indító *Local Communicator* szerver elérhetőségét. Azért nem magát a doboz elérhetőségét, mert a doboz nem szerver alkalmazás, nem lehet egyetlen függvényét sem meghívni. Az őt felügyelő *LocalComm* szerver viszont információkat tud majd róla szolgáltatni (fut-e), le tudja állítani (kill), stb.

> *TErrCode RCRegisterChannelProxy(TProjectID projectID, TTypeOfChannel typeOfChannel, TChannelID channelID, TProxyDescr channelProxy);*

A függvény segítségével értesíti a futtató rendszert a csatornapéldány, hogy sikeresen elindult, és kész teljesíteni a feladatait. A csatorna ennek során azonosítja magát a szokásos hármassal, valamint a saját elérhetőségét (*channelProxy*).

> *TErrCode RCRegisterChannel(TProjectID projectID, TBoxID boxID, TThreadID threadID, TChannelTask channelTask, TTypeOfChannel typeOfChannel, TChannelID channelID, int collection);*

Ezen függvény segítségével értesíti a doboz a futtató rendszert, hogy egy csatornapéldány indítását sikeresen befejezte, és ezen csatornát a továbbiakban használni kívánja. A doboz azonosítja magát a szokásos hármassal, majd megadja a csatorna típusát (*typeOfChannel*), és az azonosítóját (*channelID*). Megadja, milyen célra kívánja használni a csatornát (*channelTask*), illetve melyik kollekcióba tartozik ez a csatorna.

> *int RCGetChannel(TProjectID projectID, TTypeOfChannel typeOfChannel, TChannelID channelID, out TProxyDescr channelProxy);*

Ezzel a függvénnyel lehet egy csatorna elérhetőségét lekérdezni. Meg kell adni a csatorna melyik projektbe tartozik, mi a típusa, mi az azonosítója. A függvény megadja a csatorna proxy címét (*channelProxy*). Sikeres esetben a 0 (nem volt hiba) hibakóddal tér vissza, egyéb esetben a hibakóddal.

> *ArrayOfTChannelID RCGetStartedChannels(TProjectID projectID, TBoxID boxID, TThreadID threadID, TChannelTask channelTask);*

Ezzel a információs függvény segítségével lehet lekérni egy adott dobozhoz tartozó csatornákat. A dobozt a szokásos hármas azonosítja. A *channelTask* adja meg, hogy az input vagy az output csatornákat kérdezzük-e le. A függvény visszatérési értéke a csatornák azonosítóinak egy tömbje. Amennyiben a doboznak nincs csatornája (vagy maga a doboz sem ismert), a csatornák azonosítójának listája egy 0 elemű tömb lesz.

```
int RCGetStartedThreadCount( TProjectID projectID, TBoxID boxID,
  TThreadID threadID, TSubGraphTask subGraphTask );
```

Ezen függvény segítségével kérhetjük le, hogy egy adott doboz hány algráfot indított el. A dobozt a szokásos hármas azonosítja. A *subGraphTask* értéke mutatná az algráf célját, ez későbbi fejlesztésre van fenntartva.

```
TErrCode RCFinishedStartingChannels( TProjectID projectID, TBoxID boxID,
  TThreadID threadID,  TChannelTask channelTask );
```

Egy doboz ezen függvény meghívásával jelzi, hogy minden (adott *channelTask*-al rendelkező) csatorna indítását sikeresen befejezte.

```
TErrCode RCStartStartingThreads( TProjectID projectID, TBoxID boxID,
  TThreadID threadID, TSubGraphTask subGraphTask );
```

Ezen függvény segítségével jelzi egy doboz, hogy algráfok indítását kezdi el. A *subGraphTask* jelezné az indítás okát, jelenleg ez nincs használva.

```
TErrCode RCAddThreadCount( TProjectID projectID, TBoxID boxID,
  TThreadID threadID, TSubGraphTask subGraphTask,  int threadCount );
```

Ezzel a függvénnyel állíthatja be egy doboz, hogy a *startGraph* kifejezéssel hány algráfot indított el. A *subGraphTask* elvileg az egyéb okokból indítást specifikálná, jelenleg ez nincs használva. A *threadCount* érték additívan van értelmezve.

```
TErrCode RCNewThreadStarted( TProjectID projectID, TBoxID boxID,
  TThreadID threadID, TSubGraphTask subGraphTask  );
```

Ezen függvény megfelel a *RCAddThreadCount* függvényhívás hatásával, speciálisan a $threadCount = 1$ paraméter megadása mellett.

```
TErrCode RCFinishedStartingThreads( TProjectID projectID, TBoxID boxID,
  TThreadID threadID, TSubGraphTask subGraphTask );
```

Ezen függvény segítségével jelzi egy doboz, hogy algráfok indítását sikeresen befejezte, és az indított szálak darabszámát be is állította.

C.1. megjegyzés. A *RCStartStartingThreads* - *RCFinishedStartingThreads* párosra azért van szükség, hogy az indított szálak számát lekérdező függvény tudja, hogy milyen állapotban van az indítás. Amennyiben az indítás még el sem kezdődött, vagy folyamatban van, úgy várakozni kell a végleges érték kialakulására.

```
void RCRegisterLocalComm( TProxyDescr localCommLocation );
```

Ezzel a függvénnyel egy *LocalComm* szerver értesíti a rendszert, hogy fut és üzemel, megadván a saját elérhetőségét.

```
void RCRegisterAppStarter( TProxyDescr appStarterLocation );
```

Ezzel a függvénnyel egy *AppStarter* szerver értesíti a rendszert, hogy fut és üzemel, megadván a saját elérhetőségét.

```
TProxyDescr RCGetAppStarter( );
```

Ezen függvény segítségével egy *AppStarter* szerver elérhetőségét lehet lekérdezni. Ilyen szerverből egy is elég, és minden példánya elvileg egyenrangú. Amennyiben több *AppStart* is regisztrált volna a rendszerben, úgy a legutolsó regisztrációja kerül megadásra.

```
void RCRegisterCodeLibrary( TProxyDescr codeLibLocation );
```

Ezzel a függvénnyel egy *CodeLib* szerver értesíti a rendszert, hogy fut és üzemel, megadván a saját elérhetőségét.

```
ArrayOfTProxyDescr RCGetAllCodeLibs();
```

Ezen függvény segítségével az összes, korábban regisztrált *CodeLib* szerver elérhetőségét le lehet kérdezni. Mivel ezek nem ekvivalensek (egyesek más-más platformú kódokat kezelhetnek), így az összes címének letöltésére szükség lehet.

```
void RCRegisterDMonitor( TProxyDescr monitorLocation );
```

Ezzel a függvénnyel egy *Monitor* szerver értesíti a rendszert, hogy fut és üzemel, megadván a saját elérhetőségét.

```
TProxyDescr RCGetMonitor( );
```

Ezen függvény segítségével egy *Monitor* szerver elérhetőségét lehet lekérdezni. Ebből a fajta szerverből egy is elég, sőt, több példánya ha fut, akkor előfordulhat, hogy egyes komponensek az egyik monitort használják log írására, mások másikat. Amennyiben több *Monitor* is regisztrált volna a rendszerben, úgy a RegCenter mindegyiket ellenőrzi, hogy a cím még működik-e. Amennyiben nem működő címre bukkan, automatikusan eltávolítja a listájáról. Ha nem talál működő *Monitor* szervert, akkor üres stringet ad vissza.

```
ArrayOfTProxyDescr RCGetAllLocalComms( );
```

Ezen függvény segítségével az összes, a rendszerben regisztrált *LocalComm* szerver elérhetőségét lehet lekérdezni. A címeket a fogadó oldalon ellenőrízni kell, mivel a *LocalComm*-ok nem értesítik a *RegCenter*-t a leállásuk (vagy meghibásodásuk) esetén.

```
void KillAllProcess( TProjectID projectID );
```

Ezzel az függvénnyel lehet egy (vagy minden) futó projekt komponenst leállítani. Amennyiben a *projectID* helyén '*' karaktert adunk meg, úgy az minden futó komponens leállását okozza.

C.2. megjegyzés. A függvény az összes *LocalComm* -ot értesíti, hogy *kill*-t kell végrehajtani. Mivel mind a dobozok, mint a csatornák indítását valamely *LocalComm* szerver végezte, így ismeri azok operációs rendszerbeli process azonosítóit, és operációs rendszbeli szinten kezdeményezheti azok leállását.

C.3. megjegyzés. A függvény ezen felül törli a *RegCenter*-ben tárolt minden hivatkozást is az adott projekt elemeire (csatornák, kiosztott szál-azonosítók, stb). Így ha valamely projekt komponens meg is „úszta" a törlést, a következő projekt indítás során nem okozhat zavart.

C.5. Local Communicator

A *LocalComm* szerverek minden olyan számítógépen futnak, amelyek számítási feladatot is végezhetnek. Egy ilyen egységre csatornák és dobozok tölthetők le és indíthatók el. Ehhez az adott gépen futó szerver alkalmazásra van szükség, amely ezen feladatokra megkérhető. Ezen kívül további, futás közbeni hasznos funkciókra is ad támogatást.

```
void CppLock();
```

Ezen függvény egy *lock* utasításnak felel meg. Lock-ra szükség lehet egy doboz életében, de a Clean nyelv nem ad erre támogatást. Ezért ezt ezzel a külső függvénnyel lehet megoldani.

C.4. megjegyzés. Lock-ra szükség van, amikor a doboz új csatornát indít. A doboz által indított csatornák proxy leíróit a dobozhoz csatol C függvények egy listában tárolják, mivel a Clean-ből ezekre sorszámukkal hivatkozunk. Amikor új csatorna kerül hozzáadásra, a lista bővítésére kerülhet sor, amely közben egyéb műveleteket nem szabad végezni. Egy csatornaindítás után ugyanakkor aszinkron módon értesülhet a doboz a csatorna helyéről, melynek hatására a listát módosítaná, miközben annak bővítése (a lista elemek új helyre másolása) van folyamatban.

```
void CppRelease();
```

Ezen függvény a *CppLock()* függvény párja. A lock megszerzése után a csatornalista bővítése következhet, majd ezen *release* segítségével a lock-ot fel lehet szabadítani.

> *TErrCode ChannelStart(TProjectID projectID, TTypeOfChannel typeOfChannel,*
> *TBoxID boxID, TThreadID threadID, TChannelTask channelTask,*
> *TChannelID suggestedChannelID, int collection, out TChannelID newChannelID,*
> *out TProxyDescr channelProxy);*

Ezzel a függvénnyel lehet kezdeményezni egy csatorna indítását. Ennek során az indítást kérő doboz a szokásos hármassal azonosítja magát, majd megadja az indítandó csatorna típusát, feladatát (*channelTask*), ha fix azonosítójú csatornát indít akkor annak id-jét (*suggestedChannelID*) különben ide 0-t kell megadni. Megadja még melyik kollekcióba fog esni ez a csatorna. Visszakapja az indított csatorna id-jét (*newChannelID*), és az elérhetőségét (*channelProxy).*

> *TErrCode ChannelFind(TProjectID projectID, TChannelID channelStaticID,*
> *TTypeOfChannel typeOfChannel, TBoxID boxID, TThreadID threadID,*
> *TChannelTask channelTask, out TChannelID newChannelID,*
> *out TProxyDescr channelProxy);*

Ezen függvény segítségével a doboz egy fix azonosítójú csatornát deríthet fel. Ennek megfelelően paraméterezése lényegében megegyezik a *ChannelStart* függvény paraméterezésével. Azonban ha a csatorna még nem került volna indításra, akkor ez a függvény *megvárja*, amíg elindul - ellentétben a *ChannelStart*-al, amelyik el is indítja.

C.5. megjegyzés. Szokás szerint a csatornák indítása output esetben aktív tevékenység, input esetben passzív. Egy fix azonosítójú csatorna mind input mind output esetben lehetne aktív tevékenység, de hogy ne kezdeményeződjön kétszer a csatorna indítása, az a doboz, ahol a fix azonosítójú csatorna output szerepkörben van, *ChannelStart* függvényt használ, az a doboz, ahol input szerepkörű ez a fix csatorna, az pedig *ChannelFind*-t használ.

> *TErrCode SubGraphStart(TProjectID projectID, TThreadID starterThread,*
> *TBoxID starterBoxID, TSubGraphTask graphTask, TSubGraphID subgrapToStart,*
> *int count, out ArrayOfThreadID threadIDs);*

A doboz ezzel a függvényhívással kezdeményezheti egy algráf indítását. A hívás egyrészt továbbítódik egy működő *AppStarter* felé (aki a konkrét indítást levezényli), valamint a *RegCenter* felé is továbbítódik, a *RCAddThreadCount* függvény hívásával jelzi, hogy a doboz újabb algráfot indított.

> *TErrCode getBoxStartInfo(TProjectID projectID, TBoxID boxID,*
> *out TThreadID starterThreadID, out TThreadID thisThreadID, out int graphTask);*

Egy frissen indított doboz indítása után visszajelentkezik az őt indító *LocalComm* szerverre, hogy lekérje a saját azonosító adatait. Megadja saját *projectID* és *boxID* azonosítóit, és megkapja melyik szál indította őt (erre tud hivatkozni *ancestorThread* néven), mi az ő szálazonosítója (erre számtalan helyen van szüksége, minden más függvény esetén később erre hivatkoznia kell), és mi az ő algráf feladata.

```
int GetLoaded();
```

Ezt a függvényt egy *AppStarter* hívhatja meg, hogy tájékozdódjon az adott számítógép terheltségéről. A visszatérési érték (egész szám) jellemzi a terheltséget. Minél nagyobb ez az érték, annál jobban terhelt ez a számítógép.

A terheltségi érték kiszámításánál figyelembe lehet venni az elmúlt néhány perc processzorterheltségi értékét, magát a processzor sebességét, a memória kihasználtságot, stb. Jelenleg ennél egyszerűbb módon működik ez a függvény: megszámolja hány csatorna fut ezen a gépen aktuálisan és hány doboz. A csatornák 3 pontot, a dobozok 10 pontot érnek.

```
bool CapableOfRun(TPlatform platform );
```

Ezt a függvényt egy *AppStarter* hívhatja meg, átadván egy platform leíró stringet. A *LocalComm* megválaszolja, hogy az ilyen platformú kódot képes-e futtatni.

```
int DownloadAndStartChannel( ServerID serverID, TProjectID projectID,
  TTypeOfChannel typeOfChannel,  TPlatform platform, TChannelID newChannelID,
  int collection);
```

Ezt a függvényt egy *AppStarter* hívja, amennyiben úgy döntött, hogy egy csatorna indítását erre a *LocalComm* szerverre bízza. Megadja a tároló *CodeLib* szerver elérhetőségét (*serverID*), a projekt azonosítóját és a csatorna típusát, valamint a platformot. Ennyi információval már a *LocalComm* képes lekérni a szükséges fájlokat. A fájlokat lementi egy alkönyvárba (`c:\app_env\local_comm\<project_id>_<<channeltype>_<time>`), majd elindítja az .exe kiterjesztésű állományt (windows platform esetén). Parancssori paraméterként (command line) átadja az alábbi információkat: „PROJECTID=<projectID> CHANNELID=<newChannelID> COLLECTION=<collection> LC_PORT=<port>”, ahol megadja a csatorna melyik projektbe tartozik, milyen egyedi azonosítóval rendelkezik, és melyik kollekcióba tartozik. Megadja saját maga (*LocalComm*) elérhetőségét is. Mivel a csatorna ugyanezen a gépen indul el (*localhost*), így csak azt kell megadni, hogy melyik portra van telepítve.

```
TChannelID GetMyChannelID(TProjectID projectID, TTypeOfChannel typeOfChannel);
```

Egy indított csatorna ezen függvénnyel kérheti le a saját csatornaazonosítóját utólag.

C.6. megjegyzés. A csatorna ugyan parancssori paraméterben megkapja ezt az információt, de *Clean*ből nem triviális parancssori paraméterek kezelése. Ezért alternatív megoldásként ezen függvény alapján is hozzájuthat a csatorna ehhez az információhoz.

```
int DownloadAndStartNode( ServerID serverID, TProjectID projectID,
  TSubGraphID subGraphID, TPlatform platform, TBoxID boxID,
  TThreadID yourThread, TThreadID starterThreadID, TBoxID starterBoxID,
  TSubGraphTask subGraphTask);
```

Ezt a függvényt egy *AppStarter* hívja, amennyiben úgy döntött, hogy egy számítási node (doboz) indítását erre a *LocalComm* szerverre bízza. Megadja a tároló *CodeLib* szerver elérhetőségét (*serverID*), a projektazonosítóját és a dobozazonosítót, valamint a platformot. Ennyi információval már a *LocalComm* képes lekérni a szükséges fájlokat. A fájlokat lementi egy alkönyvárba (`c:\app_env\local_comm\<project_id>_<<boxid>_<time>`), majd elindítja az .exe kiterjesztésű állományt (windows platform esetén). Parancssori (command line) paraméterekként adja át az alábbi információkat:

- `PROJECTID=<projectid>`
- `BOXID=<boxid>`
- `LC_PORT=<port>`
- `ST_BOXID=<boxid>`
- `ST_THREAD=<thread>`
- `THREAD=<thread>`
- `TASK=<task>`

A paraméterek azonosítják, hogy a doboz melyik projektbe tartozik, mi az azonosítója. Megadja saját maga (*LocalComm*) elérhetőségét is. Mivel a doboz ugyanezen a gépen indul el (*localhost*), így csak azt kell megadni, hogy melyik portra van telepítve. Ezen felül megadja az indító doboz azonosítóját ($ST\_THREAD + ST\_BOXID$), és hogy ezen dobozpéldány melyik szálba tartozik ($THREAD$), és milyen taszkfeladatot lát el.

```
idempotent void KillAllProcess( TProjectID projectID );
```

Ezt a függvényt a *RegCenter* hívja, amikor utasítást kap a projekt leállítására. Miután törli a saját regisztrációs adatbázisát, minden *LocalComm* szervert is értesít a futó csatornák és dobozok operációs szintű leállításáról.

```
int SetChannelIDs( TProjectID projectID, TThreadID threadID, TBoxID boxID,
  int chnInnerID, TProxyDescr channelProxyStr );
```

A doboz, amelyik *SplitF* protokoll használatára készült, első lépésben a doboz a saját *LocalComm* szerverére feltölti az érintett csatornák elérhetőségét. A doboz a szokásos hármassal azonosítja magát, majd megadja a csatorna proxy címét, és egy belső

azonosító sorszámot. Ezt a függvényt annyiszor hívja meg, ahány csatorna szerepel a *SplitF* protokollban.

A *LocalComm* az érintett csatornákat értesíti, hogy értesítést vár tőlük a státuszuk változásakor. Megoldható lenne, hogy minden egyes alkalommal körbekérdezné a csatornákat azok szabad kapacitásáról, de teljesítménybeli megfontolások alapján inkább a csatornák jelzik, ha van szabad hely a pufferükben.

```
int SignChannelFree( TProjectID projectID, TThreadID threadID, TBoxID boxID,
  TProxyDescr chnannelPrxStr, int freeCount );
```

Ezt a függvényt egy csatorna hívja meg, jelezvén hogy változás állt be a szabad kapacitásában. Ezt az információt a csatorna csak akkor küldi automatikusan, ha a *LocalComm* ezt kérte tőle. A változás azt jelenti, hogy a puffer kihasználtsága 90% alá csökken, vagy fölé növekszik. Amikor a doboz a *SplitF* prokotokollja miatt szabad csatornára vár, ez az információ kulcsfontosságú.

```
TChannelID WaitForAnEmptyChannel( TProjectID projectID, TThreadID threadID,
  TBoxID boxID );
```

Ezt a függvényt az a doboz hívja, amelyik a *SplitF* protokollja miatt olyan csatornaazonosítót keres, amelyiken van szabad hely. A szóba jöhető csatornákat korábban a *SetChannelIDs* függvénnyel már átadta a *LocalComm*-nak. Ezen függvény a doboz futását mindaddig *blokkolja*, amíg szabad csatorna nem keletkezik. Amennyiben a megadott csatornák között van olyan, amelyik 90% alatti terheltségű, úgy a függvény ennek belső azonosítójával tér vissza (amelyet a doboz választott magának szintén a *SetChannelIDs* segítségével).

C.6. Monitor

A rendszerben helyet kap egy *Monitor* alkalmazás, amelynek üzeneteket lehet küldeni. Az üzeneteket a monitor log fájlba írja, és a képernyőn is megjelenítni. Ez igen hasznos hibakeresés és nyomkövetés esetén, az események utólag is nyomonkövethetőek és elemezhetőek.

```
void ProjectEvent( long timeStamp, TProjectID projectID,
  TProjectEventCode eventCode,  string message);
```

Projekt szintű esemény (indítási kérelem, leállítási kérelem) jelzésére használható függvény. Az esemény jellegét az *eventCode* adja meg, és csatolható hozzá egy string alakú üzenet is. Ezt jellemzően az *AppStarter* (indítás) vagy *RegCenter* (leállítás) hívja meg.

```
void BoxEvent( long timeStamp, TProjectID projectID,
 TThreadID starterThread, TBoxID BoxID, TBoxEventCode eventCode,
 string message);
```

Doboz szintű esemény (csatorna indítása, indítás befejezése, algráf indítása, feldolgozás kezdése) bekövetkeztét, az esemény valamely fázisába lépést jelezhetjük.

```
void ChannelEvent( long timeStamp, TProjectID projectID,
 TTypeOfChannel typeOfChannel, TChannelID chnID, string functionName,
 int callID, Phases phaseID, ErrFlags errFlag, string message);
```

Csatornák által használt függvény. Megadhatjuk melyik függvényét hívták meg legutóbb (*functionName*), a hívás valamely belső sorszámát (*callID*), a függvényen belüli fázis azonosítóját (*phaseID*), hibakódot és üzenetet.

C.7. megjegyzés. Ez azért ilyen részletes üzenet, mert a csatornafüggvények hajlamosak a blokkolásra. Egy kiolvasási kérelem függvényhívás (retrieve) első fázisa maga a függvény hívása, majd második fázisban blokkolásba lépés, harmadik fázisa a blokkolásból ébredés (ha elem került a pufferbe), negyedik fázisa az elem visszaadása a hívás helyére. A csatornának lehetősége van minden egyes függvényhívásakor egy belső sorszámozást használva a függvényhíváshoz azonosítót rendelni, így nyomon követhető az egyes függvényhívások milyen fázisokat jártak be, és eredményesek (sikeresek) voltak-e.

```
void ServerEvent( long timeStamp, string serverName, string functionName,
 int callID, Phases phaseID, ErrFlags errFlag, string message);
```

A futtató rendszer komponensei (*LocalComm*, *AppStarter*, stb) jelezhetik eseményeiknek a bekövetkeztét a monitor felé ezen függvény hívásával.

```
void DebugMsg( string sender, string message );
```

Bárki küldhet egy eseti nyomkövetési üzenetet ezzel a függvénnyel. Ezt használhatjuk valamely részfolyamat részletesebb nyomkövetéséhez. Amikor a hiba kiderül, és javításra kerül, a végleges változatból a *DebugMsg* függvényhívásokat el szoktuk távolítani.

```
string getProjectInfo( string projectName );
```

Ezt a függvényt használhatjuk egy projekttel kapcsolatos időmérő tevékenységhez. A függvény egy |-lal (függőleges vonallal) határolt stringet ad vissza, amely a függőleges vonal mentén darabolható az alábbi elemekre:

- `ProjectID=<project>`
- `StartRequested=<time>`

- `EarlierTime=<time>`
- `LatestTime=<time>`
- `StopRequested=<time>`
- `Comp.Nodes=<db>`
- `Channels=<db>`
- `Status=<status>`

Ezen string megadja, mikor kértük a projekt indítását (*StartRequested*), a legkorábbi, a projekttel kapcsolatos, a monitorhoz beérkező üzenet időpontját (*EarlierTime*), a legkésőbbi üzenet időpontját (*LatestTime*). E kettőből számolható a projekt futási ideje. Amennyiben kértünk leállítást, annak időpontját is megadja. Ezen felül megadja a számítási pontok (*Comp.Nodes*) darabszámát és a csatornák darabszámát, amely a projekt futása során elindult. A projekt státusza 'K'=killed, 's':started, '-':ismeretlen lehet.

```
void WriteScreen( string projectName, string message );
```

Ezzel a függvénnyel valamely csatorna vagy doboz írhat a közös képernyőre.

```
string GetScreenMessages( string projectName );
```

Ezzel a függvénnyel lekérhetjük, milyen üzeneteket írtak a képernyőre a *WriteScreen* függvény segítségével.

D. függelék

D-Clean és D-Box fordító beállításai

A D-Clean és D-Box fordító integrálva van egyetlen `DCleanComp.exe` fájlba. Parancssori paraméterek formájában kell megadni, hogy D-Clean vagy D-Box fordító működést várunk el tőle. A fordító külső XML formájú konfigurációs fájlban tárolja a működéshez szükséges beállításait. Ezek a beállítások elsősorban a D-Clean fordító számára fontosak (típuslevezetés), valamint a kódgeneráláshoz szükséges sablonfájlok neveit, illetve a használt C#, C, ICE compilerek és linkerek elérési útvonalát tárolja.

D.1. A LocalSett.XML fájl tartalma

Az XML fájlbeli tag-ek nevei a további tag-ek értékében hivatkozhatóak, `$` jelek használatával. Tehát ha pl. a *CLEAN_ROOT* tag értéke már beállításra került, akkor egy másik tag értékében szereplő `$CLEAN_ROOT$\Libraries` megjelölés alatt azt értjük, hogy a *CLEAN_ROOT* értékéhez fűzzük hozzá a `\Libraries` értéket.

Tartalmazza a telepített Clean rendszer alkönyvtárszerkezetének, és legfontosabb fájlok neveinek értékét az aktuális gépen.

- *CLEAN_ROOT*, mely a tényleges fordításnál használt Clean nyelvi rendszer gyökérkönyvtára mutat
- *CLEANLIB*, mely a Clean rendszer *LIB* alkönyvtárának neve. Ez sajnos Clean verziófüggő elnevezésű, a Clean 2.1-ben pl *Libraries*, míg Clean 2.2-ben már *Libraries 2.2* néven kerül be a Clean alkönyvtárszerkezetbe.
- *CL_COMP*, amely a Clean parancssori fordító nevét adja meg

Tartalmazza a D-Box kódgenerálás során felhasznált külső séma és sablon fájlokat tartalmazó alkönyvtárak neveit. Ezek a séma és sablon fájlok szintén változhatnak a fordító verziójának fejlődésével, de az őket tartalmazó alkönyvtárnevek valószínűleg nem változnak.

- *SCHEME* a sémák alkönyvtárja. A fordító indulásakor minden séma fájl (.scheme kiterjesztésű fájl) tartalmát automatikusan beolvassa.

- *SkelRootDir* a template fájlok alkönyvtárszerkezetének gyökere. Az alkönyvtárszerkezet e pontból kiinduló további szerkezetét a $DCleanConf.XML$ tartalma specifikálja.

D.2. A DCleanConf.XML fájl tartalma

Ez a konfigurációs fájl tartalmazza azokat a beállításokat, amelyek mellett a kódgenerálás működhet. A *CONFIG/LIBRARY* szekció tartalmazza azoknak az alkönyvtáraknak a neveit, amelyekben Clean modul definíciós (.dcl) fájlok találhatóak. Ezen fájlok tartalmának beolvasásával lehet a Clean library függvények típusinformációját elemezni, mely szükséges a D-Clean nyelvi fordításhoz, konkrétabban a típuslevezetéshez. Mivel a D-Box nyelvi szinten erre már nincs szükség, ezért ennek tartalma jelen esetben érdektelen.

A *DBox* szekció platformspecifikus beállításokat tartalmaz a kódgeneráláshoz. Minden platform esetén meg kell adni az operációs rendszer és a használt middleware nevét, valamint egy rövid azonosító nevet. Ezen azonosító névvel kerül a generált kód megjelölésre. A *LocalComm* és az *AppStarter* ezen neveket használja annak egyeztetésére, hogy az adott kódot képes-e letölteni és futtatni. Ugyanilyen nevű alkönyvtárban fogja keresni a template-ket a fordító, melyek gyökérkönyvtárának nevét a *LocalSett.XML* fájlbeli *SkelRootDir* írja le.

A kódgenerálás során az *OutputRootDir* beállítású alkönyvtárba kerülnek be a Clean, C, C# nyelvi forráskódok. Ebbe az alkönyvtárba kerülnek be a fordítást vezérlő fájlok (*makevars, makefile* is, melyek eredeti sablon forrását a*SkelRootDir* kell tartalmazza.

A platform alkönyvtárba több további alkönyvtár is bekerül majd:

- *CompNodeShared* alkönyvtárba kerülnek be azok a fájlok, melyek minden computational node (számítási csomópont, D-Box példány) fordításához szükséges fájlok.

- *CompNode* alkönyvtárba kerülnek valamely konkrét comp. node fordításához szükséges fájlok. Ezek a sablonok annyiszor kerülnek felhasználásra, ahány D-Box definíciót kell lefordítani.

- *ChannelShared* alkönyvtárba kerülnek azok a fájlok, melyek közösek a csatornák forráskódjaira nézve.

```
<?xml version="1.0"?>
<APP>
    <CONFIG currentSettings="">
     <VARIABLE name="CLEAN_ROOT" value="V:\XBIN\Clean 2.2"       />
     <VARIABLE name="CLEANLIB"    value="$CLEAN_ROOT$\Libraries" />
     <VARIABLE name="STDENV_DBG" value="$CLEANLIB$\WrapDebug" />
     <VARIABLE name="STDENV_LIB" value="$CLEANLIB$\StdEnv"/>

     <VARIABLE name="CL_COMP"
     value="$CLEAN_ROOT$\Tools\Clean System\CleanCompiler.exe" />
     <VARIABLE name="CL_CODE"
       value="$CLEAN_ROOT$\Tools\Clean System\CodeGenerator.exe" />
     <VARIABLE name="CL_LINK"
           value="$CLEAN_ROOT$\Tools\Clean System\StaticLinker.exe" />

    <VARIABLE name="SCHEME"
      value="P:\D-Clean\skeletons\schemes"       />
    <VARIABLE name="SkelRootDir"
      value="P:\D-Clean\skeletons\Skeletons"       />

    <VARIABLE name="CFORD"
      value="C:\Program Files\Microsoft Visual Studio 8\Vc\bin\cl.exe" />
    <VARIABLE name="CLINK"
      value="C:\Program Files\Microsoft Visual Studio 8\Vc\bin\link.exe" />
    <VARIABLE name="CINCLUDE1"
      value="C:\Program Files\Microsoft Visual Studio 8\Vc\PlatformSDK\Include" />
    <VARIABLE name="CINCLUDE2"
      value="C:\Program Files\Microsoft Visual Studio 8\Vc\Include" />
    <VARIABLE name="VC7_CLIB"
      value="C:\Program Files\Microsoft Visual Studio 8\Vc\lib"              />
    <VARIABLE name="PFRM_CLIB"
      value="C:\Program Files\Microsoft Visual Studio 8\Vc\PlatformSDK\Lib" />

    <VARIABLE name="C#FORD"
      value="C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\csc.exe" />
    <VARIABLE name="ICE"
      value="v:\xbin\ice-3.2.1"       />

    </CONFIG>
</APP>
```

D.1. példa. LocalSett.XML lehetséges tartalma

- *Channel* alkönyvtárba kerülnek a clean dobozok és az adott típusú csatornák közötti illesztés kódjai. Mivel ezek az adott típusú csatornák műveleteit meghívó, C-ben kódolt Clean interface-szel ellátott kódok, így annyi alkönyvtár képződik majd, ahány különböző típusú csatornát tartalmaz a D-Box projekt.

- *ServerShared* az egyes csatornák generálása során felhasználható, a különböző típusú csatornák által használt közös forráskódok.

- *Server* alkönyvtárba kerülnek az egyes csatornák forráskódjai. Ilyen alkönyvtárból majd annyi lesz, ahány különböző típusú csatornát egy konkrét D-Box projekt tartalmaz.

Az egyes alkönyvtárakbeli tartalom, a bennük levő fájlok száma és jellege az egyes platformok esetén eltérő lehet. Ezért minden, a fentiekben leírt bejegyzés számos fájlnevet tartalmaz, mely fájlokat három dolog jellemez:

- *source* a hasonló nevű template alkönyvtárakban szereplő fájlnak a neve,

- *target* a D-Box kódgenerálás utáni fájlnév, mely úgy keletkezik, hogy a template fájl tartalmát a kódgenerálás közben beolvassuk, a konkrét típusok alapján elvégezzük a fájl tartalmának átalakítását, és az eredményt ilyen néven mentjük el,

- *typeof* a fájl tartalmát jellemző adat (lásd alább).

Az egyes template fájlok tartalma eltérő lehet. Vannak olyan fájlok, melyek bináris tartalma (.obj fájl) a kódgenerálás közben nem változhat meg, ezeket csak egyszerűen át kell másolni. A bináris fájlok a *file* xml tag-en belüli *OBJ* típussal kerülnek megjelölésre. A nem bináris fájlok tartalmát elemezni kell soronként, és elvégezni a megfelelő átalakításokat. Amennyiben a fájl tartalma nem bináris, úgy altípusa (c++, clean, c# forráskód, make file) a jelenlegi implementációban nem okoz különbséget, későbbi fejlesztési okokból különböztetjük meg a típusukat.

A file nevekben használható két makró jellegű név:

- $BOXID$: az aktuálisan generált D-Box doboz azonosítója,

- $TYPENAME$: az aktuálisan generált csatorna típusneve.

A *code* tag speciális jellemzőjű. A tényleges, hibátlan fordítási folyamat végén keletkeznek az adott nyelvi (Clean, C#) compiler és linker működésének eredményeképp. Ezeket a fájlokat kell majd *CodeLib* szerverre felmásolni.

Az *obj* tag szintén speciális jellemzőjű. A Clean nyelvi kód szerkesztéséhez van szükség ezen fájlokra, melyeknek neveit a linkernek át kell adni. Ezek az *obj* fájlok tehát nem léteznek a template alkönyvtárban, hanem a sikeres fordítási folyamat végén keletkeznek. Ugyanakkor ezen bináris fájlokat nem kell feltölteni a *CodeLib* szerverre.

```
<APP>
 <CONFIG>
  <LIBRARY>
            <lib directory="$CLEANLIB$\StdEnv" />
            <lib directory="$CLEANLIB$\ArgEnvWindows" />
            <lib directory="$CLEANLIB$\Directory" />
            <lib directory="$CLEANLIB$\Dynamics" />
            <lib directory="$CLEANLIB$\ExceptionsWindows" />
            <lib directory="$CLEANLIB$\ExtendedArith" />
            <lib directory="$CLEANLIB$\GameLib" />
            <lib directory="$CLEANLIB$\Generics" />
            <lib directory="$CLEANLIB$\IOInterface" />
            <lib directory="$CLEANLIB$\MersenneTwister" />
            <lib directory="$CLEANLIB$\ObjectIO" />
            <lib directory="$CLEANLIB$\ObjectIO\OS Windows" />
            <lib directory="$CLEANLIB$\StdEnv Sparkle" />
            <lib directory="$CLEANLIB$\StdLib" />
            <lib directory="$CLEANLIB$\Tcp" />
            <lib directory="$CLEANLIB$\WrapDebug" />
            <lib directory="..\Src_Additional" />
            <lib2 directory="$CLEANLIB$\Hilde" />
            <lib2 directory="$CLEANLIB$\Dynamics\extension" />
            <lib2 directory="$CLEANLIB$\Dynamics\implementation" />
      </LIBRARY>
 </CONFIG>
 <DBox>
  ....
 </DBox>
</APP>
```

D.2. példa. DCleanConf.XML-beli LIBRARY szekció lehetséges tartalma

```
<APP>
 <CONFIG>
  ...
 </CONFIG>
 <DBox>
     <Platform OperatingSystem="Windows" Middleware="ICE" name="WIN_ICE">
      <OutputRootDir location="WINDOWS_ICE" />
          <FILE source="makevars" target="makevars" typeof="MAKE"/>
          <FILE source="Makefile" target="Makefile" typeof="MAKE"/>
          <FILE source="uploads" target="uploads" typeof="MAKE"/>

          <!-- Comp.node SHARED files -->
          <CompNodeShared sourcedir="">
               <FILE source="Box_SHARED/DClean.icl"
                    target="DClean.ICL" typeof="Clean_ICL"/>
               <FILE source="Box_SHARED/DClean.dcl"
                    target="DClean.DCL" typeof="Clean_DCL"/>
               <FILE source="Box_SHARED/DCleanDefs.dcl"
                    target="DCleanDefs.DCL" typeof="TEXT"/>
               <FILE source="Box_SHARED/DCleanDefs.icl"
                    target="DCleanDefs.ICL" typeof="TEXT"/>
               <FILE source="Box_SHARED/Makefile"
                    target="Makefile" typeof="MAKE"/>
               <OBJ name="DClean.o" />
          </CompNodeShared>

          <!-- Comp.node files -->
          <CompNode>
               <FILE source="Win_ICE/BoxID_nnnn/Makefile"
                    target="Makefile" typeof="MAKE"/>
               <FILE source="Win_ICE/BoxID_nnnn/clink.lst"
                    target="clink.lst" typeof="TEXT"/>
               <FILE source="Win_ICE/BoxID_nnnn/options.o"
                    target="options.o" typeof="OBJ"/>
               <CODE name="Box_$BOXID$.exe" />
          </CompNode>
          ...
 </DBox>
</APP>
```

D.3. példa. DCleanConf.XML-beli DBox szekció lehetséges tartalma (1)

```
<APP>
 <CONFIG>
  ...
 </CONFIG>
 <DBox>
     <Platform OperatingSystem="Windows" Middleware="ICE" name="WIN_ICE">
      ...
         <!-- Channel CLIENT SHARED files -->
         <ChannelShared sourcedir="">
             <FILE source="Win_ICE/Chn_SHARED/CleanShared.h"
                 target="CleanShared.h" typeof="CPP" />
             <FILE source="Win_ICE/Chn_SHARED/CleanShared.cpp"
                 target="CleanShared.cpp" typeof="CPP" />
             <FILE source="Win_ICE/Chn_SHARED/ChannelHandling.h"
                 target="ChannelHandling.h" typeof="CPP" />
             <FILE source="Win_ICE/Chn_SHARED/ChannelHandling.cpp"
                 target="ChannelHandling.cpp" typeof="CPP" />
             <FILE source="Win_ICE/SLICE/AppEnv.ICE"
                 target="LocalComm.ICE" typeof="SLICE" />
             <FILE source="Win_ICE/SLICE/Channel.ICE"
                 target="Channel.ICE" typeof="SLICE" />
             <FILE source="Win_ICE/Chn_SHARED/Middleware.icl"
                 target="Middleware.ICL" typeof="Clean_ICL"/>
             <FILE source="Win_ICE/Chn_SHARED/Middleware.dcl"
                 target="Middleware.DCL" typeof="Clean_DCL"/>
             <FILE source="Win_ICE/Chn_SHARED/Makefile"
                 target="Makefile" typeof="MAKE" />
             <FILE source="Win_ICE/Chn_SHARED/Clean.h"
                 target="Clean.h" typeof="CPP" />
             <OBJ name="Middleware.o" />
             <OBJ name="Channel.obj" />
             <OBJ name="ChannelHandling.obj" />
             <OBJ name="CleanShared.obj" />
             <OBJ name="LocalComm.obj" />
         </ChannelShared>
         ...
 </DBox>
</APP>
```

D.4. példa. DCleanConf.XML-beli DBox szekció lehetséges tartalma (2)

```
<APP>
 <CONFIG>
  ...
 </CONFIG>
 <DBox>
      <Platform OperatingSystem="Windows" Middleware="ICE" name="WIN_ICE">
        ...
            <!-- Channel CLIENT type-dependent files -->
            <Channel sourcedir="">
                  <FILE source="Win_ICE/Chn_mmm/CleanSide.cpp"
                        target="$TYPENAME$_CleanSide.cpp" typeof="CPP" />
                  <FILE source="Win_ICE/Chn_mmm/DComm_xxx.DCL"
                        target="DComm_$TYPENAME$.DCL" typeof="Clean_DCL" />
                  <FILE source="Win_ICE/Chn_mmm/DComm_xxx.ICL"
                        target="DComm_$TYPENAME$.ICL" typeof="Clean_ICL" />
                  <FILE source="Win_ICE/Chn_mmm/Makefile"
                        target="Makefile" typeof="MAKE" />
                  <FILE source="Win_ICE/SLICE/ChannelNNN.ICE"
                        target="Channel$TYPENAME$.ICE" typeof="SLICE" />
                  <OBJ name="Channel$TYPENAME$.obj" />
                  <OBJ name="$TYPENAME$_CleanSide.obj" />
                  <OBJ name="DComm_$TYPENAME$.o" />
            </Channel>
 </DBox>
</APP>
```

D.5. példa. DCleanConf.XML-beli DBox szekció lehetséges tartalma (3)

```
<APP>
 <CONFIG>
  ...
 </CONFIG>
 <DBox>
      <Platform OperatingSystem="Windows" Middleware="ICE" name="WIN_ICE">
        ...
            <!-- Channel SERVER SHARED files -->
            <ServerShared sourcedir="ChServer_SHARED">
            </ServerShared>

            <!-- Channel SERVER type-dependent files -->
            <Server sourcedir="">
                  <FILE source="Win_ICE/SLICE/Channel.ICE"
                        target="Channel.ICE" typeof="SLICE" />
                  <FILE source="Win_ICE/SLICE/AppEnv.ICE"
                        target="AppEnv.ICE" typeof="SLICE" />
                  <FILE source="Win_ICE/Chn_Server_mmm/ChannelServer.cs"
                        target="ChannelServer.cs" typeof="CSharp" />
                  <FILE source="Win_ICE/Chn_Server_mmm/Makefile"
                        target="Makefile" typeof="MAKE" />
                  <CODE name="ChannelServer_$TYPENAME$.exe" />
            </Server>
            ...
      </Platform>
 </DBox>
</APP>
```

D.6. példa. DCleanConf.XML lehetséges tartalma (4)

D.3. Template fájlokbeli makrók

A sablonfájlok tartalmában több makrónév is található. Ilyenek a doboz kódjának linkeléshez szükséges fájllista (.lst), amely többek között a szükséges tárgykódú fájlok neveit sorolja fel és a dobozt fordító *Makefile* sablon, a dobozhoz tartozó C nyelvű, Clean nyelvű forráskódok.

Minden makró esetén megadjuk a használat (egy lehetséges) helyét, a használat módját, és az eredményére egy példát.

- `<CLEANLIBS>`ezt a paramétert a *Makefile*-ba rakhatjuk, ahol a Clean Compiler parancssori paramétereként adhatjuk át. Ez nem más, mint a DCleanConf.XML konfigurációs file APP/CONFIG/LIBRARY szekciójában szereplő *lib* bejegyzések pontosvesszővel tagolt sorozata.
  Használata: `CL_COMP_PARM=-P "<CLEANLIBS>;.;"`
  Eredménye: `CL_COMP_PARM=-P "D:/Clean/Libraries/StdEnv;`
  `D:/Clean/Libraries/StdLib;`

- `<INC_DIR_LIST>` ezt a paramétert a *Makefile*-ba rakhatjuk, ahol a Clean Compiler parancssori paramétereként adhatjuk át. Ez nem más, mint a doboz fordításához szükséges további generált kódokat tartalmazó alkönyvtárak felsorolása. Ez függ attól, hogy a doboz milyen típusú csatornákat használ input vagy output célokra, mely alkönyvtárakba tartozó csatornakezelő interface függvényeket kell importálnia.
  Használata: `CL_COMP_PARM=-P "<INC_DIR_LIST>"`
  Eredménye: `CL_COMP_PARM=-P "../Chn_SHARED;../Chn_Int"`

- `<BOXID>` annak a doboznak az azonosítója, amely forráskódjának generálása éppen folyamatban van. Szintén a *Makefile*-ba kerülhet.
  Használata: `@$(CL_COMP) $(CL_COMP_PARM) -ou <BOXID>.icl"`
  Eredménye: `@$(CL_COMP) $(CL_COMP_PARM) -ou BoxID_1000.icl"`

- `$PFRM_CLIB$` a LocalSett.XML ugyanezen nevű bejegyzésének értéke. Ez a PlatformSDK könyvtárának a neve. Például a C linkernek szóló linkelési listába kerülhet be.
  Használata: `@$(CXXLINK) @clink.lst /LIBPATH:"$PFRM_CLIB$"`
  Eredménye: `@$(CXXLINK) @clink.lst`
  `/LIBPATH:"C:\Program Files\Microsoft Visual Studio 8`
  `\Vc\PlatformSDK\Lib"`

- `$ICE$` a LocalSett.XML ugyanezen nevű bejegyzésének értéke. Ez az ICE telepítés gyökérkönyvtárának a neve. Például a C linkernek szóló fájllistába kerülhet.

Használata: `$ICE$\lib\ice.lib`
Eredménye: `d:\bin\ice-3.2.1\lib\ice.lib`

- `<CHNSHARED_OBJS>` A csatornák közötti közös kódok tárgykódú fájlistája, újsor (0D,0A karakterrel) elválasztott listája. Szintén a C linker fájllistájába kerülhet.
  Használata: `<CHNSHARED_OBJS>`
  Eredménye:
  ```
  "..\Chn_SHARED\Midleware.o"
  "..\Chn_SHARED\Channel.obj"
  "..\Chn_SHARED\ChannelHandling.obj"
  "..\Chn_SHARED\CleanShared.obj"
  "..\Chn_SHARED\LocalComm.obj"
  ```

- `<BOXSHARED_OBJS>` A dobozok számára készült (dobozfüggetlen) kódok tárgykódú fájlistája, újsor (0D,0A karakterrel) elválasztott listája. Szintén a C linker fájllistájába kerülhet be.
  Használata: `<BOXSHARED_OBJS>`
  Eredménye:
  ```
  "..\Box_SHARED\DCleanDefs.o"
  "..\Box_SHARED\DClean.o"
  "..\Box_SHARED\DClean2.o"
  ```

- `<CHANNEL_OBJS>` Az adott doboz protokolljaiban szereplő típusú csatornák kezelését végző interface fájlok listája. A C linker fájllistájába kerülhet be.
  Használata: `<CHANNEL_OBJS>`
  Eredménye:
  ```
  "..\Chn_Int\ChannelInt.obj"
  "..\Chn_Int\Int_CleanSide.obj"
  "..\Chn_Int\DComm_Int.o"
  ```

- `<CHANNEL_OBJS_ALL>` A projektben szereplő (minden) típusú csatornák kezelését végző interface fájlok listája. A C linker fájllistájába kerülhet be.
  Használata: `<CHANNEL_OBJS_ALL>`
  Eredménye:
  ```
  "..\Chn_Int\ChannelInt.obj"
  "..\Chn_Int\Int_CleanSide.obj"
  "..\Chn_Int\DComm_Int.o"
  "..\Chn_Int\ChannelReal.obj"
  "..\Chn_Int\Real_CleanSide.obj"
  "..\Chn_Int\DComm_Real.o"
  ```

- `<OBJ_FILELIST>` Egyéb, a projekthez csatolandó tárgykódú programok listája. Ezt lehet a D-Box compiler +@ parancssori paraméterével beállítani.
  Használata: `<OBJ_FILELIST>`

Eredménye: `"C:\myCode\myOwnCode.o"`
`"C:\myCode\myOtherCode.o"`

- `<TYPENAME>` Ez az adott csatorna alaptípusának a neve (pl. Int, Real, ...). Bármely séma fájlban előfordulhat.
  Használata: `#include "Channel<TYPENAME>.h"`
  Eredménye: `#include "ChannelInt.h"`

- `<CHN_DATA_IN_CPARAMLIST>` Az adott típusú csatorna esetén képzi a C nyelvű (függvények számára készülő) paraméterlistát. Ez egyszerű típus (pl *Int*) esetén ez *int data* alakú. Összetett (rekord) típus esetén annyi paraméter generálódik be a makró helyre, ahány egyszerű típusú mezőből áll a rekord. Ez a makró a *store* C nyelvi függvényben használható.
  Használata: `int <TYPENAME>_store(int channel,`
  `<CHN_DATA_IN_CPARAMLIST>)`
  Eredménye: `int MyRec_store(int channel, int field_0, int field_1)`

- `<CREATE_RECORD_FROM_PARAMS>`Az előző makróval együttműködve használható ez a makró. Amennyiben a C nyelvi függvény esetén a típus egyszerű típus, a paraméter neve *data* lesz. Amennyiben összetett rekordtípus, úgy a paraméterek neve *field_xx* alakú. Ez esetben a *data* változó ebből C nyelvi rekord konstruktor segítségével készítendő el. Tehát egyszerű esetben ezen makró helyére nem helyettesítődik be semmi, rekord típus esetén pedig a *data* nevű változó konstruálását tartalmazza. Ez azért szükséges, hogy a szóban forgó C nyelvi függvény törzsében a továbbiakban már egységesen a *data* változóra lehessen hivatkozni.
  Használata: `<CREATE_RECORD_FROM_PARAMS>`
  Eredménye: `MyRec data = {field_0, field_1};`

- `<CHN_DATA_OUT_CPARAMLIST>`A *restore* csatorna kezelő függvény visszatérési értékeként a hibakódot adja meg (*int*), a csatornáról olvasott tényleges adatokat kimenő paraméterekken keresztül lehet leolvasni. Ez egyszerű típusú csatorna esetén egyetlen paraméter, rekord típusú csatorna esetén annyi paramétert jelent, ahány egyszerű mezőből a rekord ténylegesen áll.
  Használata: `int <TYPENAME>_retrieve(int channel,`
  `<CHN_DATA_OUT_CPARAMLIST>)`
  Eredménye: `int MyRec_retrieve(int channel, int* field_0,`
  `int* field_1)`

- `<CPPTYPENAME>` ezen makró helyére szúródik be az adott csatorna típus C nyelvi típusneve. Egyszerű típus esetén a megfelelő C nyelvi típusnév, rekord esetén a rekord neve.

Használata: `<CPPTYPENAME> data;`
Eredménye: `MyRec data;`

- `<FILL_PARAMS_FROM_CHN_DATA>` A makrót szintén a *retrieve* függvényben használhatjuk, amennyiben rekord típusú a csatornánk. Ez esetben a *data* nevű rekord típusú változó mezőiből egyesével kell kimásolni az értékeket a retrieve függvény kimenő paramétereibe.
  Használata: `<FILL_PARAMS_FROM_CHN_DATA>`
  Eredménye:
  ```
  *field_0 = data.field0;
  *field_1 = data.field1;
  ```

- `#<ifdef> STRUCTURE_BASED` Ezen feltételes makró törzs részében lévő sorok csak akkor kerülnek be a forráskódba, ha az aktuálisan generált csatorna séma alapja nem egyszerű, hanem rekord típus. Ehhez a feltételes makróhoz `#<else>` és `#<endif>` makrók is tartoznak. Az *#else* makró használata nem kötelező. A törzs részekben további makrók is szerepelhetnek.
  Használata:

  ```
  #<ifdef> STRUCTURE_BASED
  <CPPTYPENAME> data;
  status = chn->retrieve(0,data);
  #<else>
  status = chn->retrieve(0,*data);
  #<endif>
  ```

  Eredménye:

  ```
  MyRec data;
  status = chn->retrieve(0,data);
  ```

- `<DCOMM_MODULE_LIST>` Ezen makró helyébe kerül a D-Clean által generált Clean nyelvi típusfüggő modulok listája.
  Használata: `import StdEnv,<DCOMM_MODULE_LIST>`
  Eredménye: `import StdEnv,DComm_MyRec,DComm_Int`

- `<LOCAL_STRUCT_DEFS>`Amennyiben valamely csatorna rekord típusú, úgy ezen makró segítségével lehet a Clean nyelvi rekorddefiníciót generáltatni.
  Használata: `<LOCAL_STRUCT_DEFS>`
  Eredménye:

  ```
  ::MyRec={
  rec_x::Int,
  rec_y::Int}
  ```

- `<TRANSMISSIBLE_TYPES>` Ezen makró helyére kerülnek be a projektben szereplő csatornatípusokhoz tartozó Clean nyelvi típuskonstruktorok definíciói.
  Használata: `:: Transmissible = <TRANSMISSIBLE_TYPES>`
  Eredménye:
  ```
  :: Transmissible = T0E  MyRec| T0L  [MyRec]|
                     T0LL [[MyRec]]| T0LLL [[[MyRec]]] | T1E  Int|
                     T1L  [Int]| T1LL [[Int]]| T1LLL [[[Int]]]
  ```

- `<F_TYPEID>` Ezen makró helyére adott típus esetén a *transmissible* típuskonstruktorának neve szúródik be.
  Használata: `<F_TYPEID>E  i -> SplitF_ [[i]] (map makeChannel chns) w`
  Eredménye: `T0E  i -> SplitF_ [[i]] (map makeChannel chns) w`

- `<F_TYPENAME>`Ezen makró jelöli az aktuális csatorna Clean típusának nevét.
  Használata: `getData<F_TYPEID>L  :: Transmissible -> [<F_TYPENAME>]`
  Eredménye: `getDataT0L  :: Transmissible -> [MyRec]`

- `#<foreach_types>` .. `#<end_foreach_types>`Ezen makró törzsében szereplő sorok annyiszor szúródnak be, ahány különböző csatornatípust használunk. Az egyes típusok esetén működnek a `<F_TYPEID>`, a `<F_TYPENAME>` makrók.
  Használata:

  ```
  #<foreach_types>
  getData<F_TYPEID>E   :: Transmissible -> <F_TYPENAME>
  getData<F_TYPEID>L   :: Transmissible -> [<F_TYPENAME>]
  getData<F_TYPEID>LL  :: Transmissible -> [[<F_TYPENAME>]]
  getData<F_TYPEID>LLL :: Transmissible -> [[[<F_TYPENAME>]]]
  #<end_foreach_types>
  ```

  Eredménye:

  ```
  getDataT0E   :: Transmissible -> MyRec
  getDataT0L   :: Transmissible -> [MyRec]
  getDataT0LL  :: Transmissible -> [[MyRec]]
  getDataT0LLL :: Transmissible -> [[[MyRec]]]
  ```

D.4. DClean fordító paraméterezése

A D-Clean, D-Box compiler egyetlen futtatható fájlból áll: *DCleanComp.exe*, mely két konfigurációs fájlból dolgozik. A futtatható program parancssori működésű, a paramétereit parancssorban kell neki átadni. A paraméterek lehetséges értékét `/?` paraméterrel is le lehet kérni. A legfontosabb paraméterek jelentését az alábbiakban adjuk meg:

- `-WDBOX` ezen paraméter jelenléte esetén a fordító *D-Box* üzemmódban működik, hiányában *D-Clean* fordítóként (alapértelmezett viselkedés).

- `programnev.icl` a parancssori paraméterek egyik pozícióján állhat egy *.icl* kiterjesztésű fájl neve. Ezt a fordító automatikusan Clean nyelvi forráskódnak értékeli. Ha *D-Clean* fordítóként használjuk, akkor ebben a fájlban keresi a *DistrStart* kifejezést is. Amennyiben *D-Box* fordítóként használjuk, úgy ezt a fájlt használja kódgeneráláshoz, ebben feltételezi a felhasználó által definiált függvények létét.

- `programnev.box` a parancssori paraméterek valamely pozícióján álló *.box* kiterjesztésű fájlt a fordító szintén kétféleképpen képes értelmezni. Amennyiben a *D-Clean* üzemmódban van, a *DistrStart* kifejezés által leírt számítási gráf alapján generálja a D-Box nyelvi definíciókat, és beleírja ebbe a fájlba (output). Ha D-Box üzemmódban használjuk a fordítót, akkor ebből a fájlból olvassa ki a D-Box nyelvi definíciókat.

- `-P projektnév` kapcsolóval kell megadni a generált projekt azonosítóját. Ezen azonosító bekerül minden generált doboz és csatorna forráskódjába, és a futtató rendszerben is hivatkozni kell majd ezen azonosítóra, ha műveleteket kívánunk a projekttel végezni (indítás, leállítás, státuszlekérdezés, stb). Amennyiben nem adunk meg projekt nevet, úgy a fordító egy random betűsorozatot generál.

- `+@ fileName`, ahol a fájlnév egy text fájlt azonosít. Ezen fájlban soronként egy tárgykódú (object) fájl neve szerepel teljes elérési úttal megadva. Ezek a sorok a szerkesztő (linker) számára készített lista fájlba kerülnek be, ily módon hozzá-szerkesztődnek a dobozok futtatható állományához.

- `-T dirname` paraméter segítségével adhatjuk meg, melyik alkönyvtárba helyezze a fordító a generált fájlokat. Alapértelmezett alkönyvtár a *./GENERATED*.

E. függelék

AppEnv segédprogram

Az *AppEnv* futtató rendszer működtetéséhez egy parancssori eszközt készítettünk el (`XAppEnv.exe`). A program a parancsokat minden esetben a *RegCenter* felé továbbítja, ezért szükséges egy elérhető *RegCenter*nek lenni az alhálózatban. Az *XAppEnv* program indulásakor megvárja, amíg ez a *RegCenter* broadcast üzenete jelzi annak helyét. A program az alábbi parancssori paramétereket tudja kezelni:

- `@ fájlnév` batch üzemű végrehajtás: a további parancsok soronként kerülnek a megadott text fájlból feldolgozásra,
- `-P projektnév` megadhatjuk melyik projekttel kívánunk műveletet végezni,
- `-EMPTY` kezdeményezzük az adott projekthez tartozó fájlok törlését a *CodeLib* szerverekről,
- `-START` kezdeményezzük az adott projekt indítását,
- `-STOP` kezdeményezzük az adott projekt leállását,
- `-STATUS` kiíródik az adott projekt státusza,
  - `-ALL` minden információ kiírása, nem csak összefoglaló,
  - `-DETAILED` a közös képernyőre írt sorok is,
  - `-CONT` 3 másodpercenként megismétel kiírás
- `-LIST` kilistázódik jelenleg milyen szolgáltatások vannak regisztrálva a *RegCenter*be,
- `-CHANNEL` csatorna fájl feltöltése valamely *CodeLib* szerverre,
  - `-TYPE típusnév` a csatorna típusa,
  - `-PLATFORM platformnév` a platform azonosító string,
  - `-FILE fájlnév` a feltöltendő csatornához tartozó fájl neve,
- `-COMPNODE` doboz fájl feltöltése valamely *CodeLib* szerverre,
  - `-SUBGRAPH szám` a doboz algráf azonosítója,
  - `-BOXID boxid` a doboz azonosítója,
  - `-PLATFORM platformnév` a platform azonosító string,
  - `-FILE fájlnév` a feltöltendő dobozhoz tartozó fájl neve,

F. függelék

D-Clean → D-Box példák

A D-Clean nyelvi struktúráknak D-Box nyelvi leírások felelnek meg. Az alábbiakban részletes ismertetés nélkül példákon keresztül mutatjuk be, milyen D-Clean kulcsszónak milyen D-Box definíció felel meg. A D-Clean nyelv részletes ismertetését a publikációs lista, illetve Zsók Viktória készülő PhD disszertációja dolgozza fel.

F.1. DStart

Clean nyelvi kód

```
1 generator:: [Int] [[Real]]
2 DistrStart = .... DStart generator
```

F.1. példa. generator függvény Clean nyelvi definíciója

D-Box definíció

```
1 BOX BoxID_1000  // Start doboz
2 {   1,
3     { ( null ), memory},
4     { ( null ), generator ,([Int],[[Real]]) },
5     { (([Int],1001, ([[Real]],1002)), split1}
6 }
```

F.2. példa. DStart nyelvi elem fordítása

F.2. DStop

Clean nyelvi kód

```
1 writeResult:: [Real] [[Int]] *World -> *World
2 DistrStart = DStop writeResult ...
```

F.3. példa. DStop nyelvi elem

D-Box definíció

```
1 BOX BoxID_1000  // Stop doboz
2 {   1,
3     { ( ([Real], 1001), ([[Int]],1002) ), join1 },
4     { ( [Real], [[Int]], *World ), writeResult ,(*World) },
5     { (( null ), memory}
6 }
```

F.4. példa. DStop nyelvi elem fordítása

F.3. DDivideS

Clean nyelvi kód

```
1 splitter:: [Real] -> [[Real]]
2 DistrStart = ... DDivideS splitter 2 ...
```

F.5. példa. DDivideS nyelvi elem

D-Box definíció

```
1 BOX BoxID_1000  // DDivideS doboz
2 {   1,
3     { (([Real], 1001)), join1 },
4     { ( [Real] ), splitter ,([[Real]]) },
5     { ( [Real], 1002), ([Real],1003), splitk }
6 }
```

F.6. példa. DDivideS nyelvi elem fordítása

F.4. DDivideN

Clean nyelvi kód

```
1 splitter:: [Real] -> [[Real]]
2 DistrStart = ... DDivideN splitter ...
```

F.7. példa. DDivideN nyelvi elem

D-Box definíció

```
1 BOX BoxID_1000 // DDivideN doboz
2 {   1,
3     { (([Real], 1001)), join1 },
4     { ( [Real] ), splitter ,([[Real]]) },
5     { autoConnBox 2 BoxID_1001 ( [Real] ), splitk }
6 }
```

F.8. példa. DDivideN nyelvi elem fordítása

F.5. DDivideF

Clean nyelvi kód

```
1 splitter:: [Real] -> [[Real]]
2 DistrStart = ... DDivideF splitter 2 ...
```

F.9. példa. DDivideS nyelvi elem

D-Box definíció

```
1 BOX BoxID_1000 // DDivideF doboz
2 {   1,
3     { (([Real], 1001)), join1 },
4     { ( [Real] ), splitter ,([[Real]]) },
5     { ( [[Real]], 1002), ([[Real]],1003), splitk }
6 }
```

F.10. példa. DDivideF nyelvi elem fordítása

F.6. DApply

Clean nyelvi kód

```
1 pickUpEvens::[Int]->[Int]
2 DistrStart = ... DApply pickUpEvens ....
```

F.11. példa. DApply nyelvi elem

D-Box definíció

```
1 BOX BoxID_1002        // DApply
2 {
3         1,
4         { (([Int],1002)), join1},
5         { ([Int]), 'pickUpEvens' ,([Int]) },
6         { (([Int],1004)), split1}
7 }
```

F.12. példa. DApply nyelvi elem fordítása

F.7. DMerge

Clean nyelvi kód

```
1 merger:: [[Int]]->[Int]
2 DistrStart = ... DMerge merger ....
```

F.13. példa. DDivideS nyelvi elem

D-Box definíció

```
1 BOX BoxID_1004        // DMerge
2 {       1,
3         { (([Int],1004),([Int],1005)), joink},
4         { ([[Int]]), 'merger' ,([Int]) },
5         { (([Int],1006)), split1}
6 }
```

F.14. példa. DDivideS nyelvi elem fordítása

G. függelék

N vezér probléma

Az alábbiakban bemutatjuk a 6. fejezetben megadott probléma D-Box nyelvi kódját.

```
typeDef Vezer = {
        Int,
        Int,
        Int }

BOX BoxID_1000      // DStart
{
    1,
    { ( null ), memory},
    { ( null ), start_gen ,([[Vezer]]) },
    { (([[Vezer]],1001)), split1}
}
BOX BoxID_1001      // DDivideF
{
    1,
    { (([[Vezer]],1001)), join1},
    { ([[Vezer]]), divider ,([[Vezer]]) },
    { (([[Vezer]],1002),([[Vezer]],1003),([[Vezer]],1004),([[Vezer]],1005)), splitf 4}
}
BOX BoxID_1002      // DApply
{
    1,
    { (([[Vezer]],1002)), join1},
    { ([[Vezer]]), (worker) ,(Int) },
    { ((Int,1006)), split1}
}
```

```
 BOX BoxID_1003       // DApply
{
       1,
       { (([[Vezer]],1003)), join1},
       { ([[Vezer]]), (worker) ,(Int) },
       { ((Int,1007)), split1}
}
BOX BoxID_1004       // DApply
{
       1,
       { (([[Vezer]],1004)), join1},
       { ([[Vezer]]), (worker) ,(Int) },
       { ((Int,1008)), split1}
}
BOX BoxID_1005       // DApply
{
       1,
       { (([[Vezer]],1005)), join1},
       { ([[Vezer]]), (worker) ,(Int) },
       { ((Int,1009)), split1}
}
BOX BoxID_1006       // DMerge
{
       1,
       { ((Int,1006),(Int,1007),(Int,1008),(Int,1009)), joink},
       { ([Int]), merger ,(Int) },
       { ((Int,1010)), split1}
}
BOX BoxID_1007       // DStop
{
       1,
       { ((Int,1010)), join1},
       { (Int,*World), saver ,(*World) },
       { ( null ), memory}
}
```